A Study of High Performance Multiple-Precision Integer Multiplication and Division Using SIMD Instructions

March 2023

Takuya Edamatsu

A Study of High Performance Multiple-Precision Integer Multiplication and Division Using SIMD Instructions

Graduate School of Science and Technology

Degree Programs in Systems and Information Engineering

University of Tsukuba

March 2023

Takuya Edamatsu

Abstract

Arithmetic operations are the most basic calculations and are versatile enough to be used in all fields. Among arithmetic operations, multiplication and division are relatively expensive due to their complexity. Modern central processing units can quickly perform 64-bit arithmetic operations, even division. However, higher-precision arithmetic operations cannot be performed with a single instruction. Instead, they must be solved in software. These operations are most commonly used in cryptography and computer algebra processing.

Single instruction, multiple data (SIMD) instructions process multiple data in parallel with a single instruction. Most modern processors support 128- and 256-bit-wide SIMD instructions. The latest processors support 512-bit-wide instructions, which allows, for example, eight 64-bit integers to be processed simultaneously. Thus, SIMD instructions have the potential to process large integer operations at high speed.

However, current SIMD instructions do not hold the carry that occurs when addition is performed. Large integer multiplication and division result in many addition operations, which generate many carries. Therefore, the carry problem cannot be ignored when using SIMD instructions. One way to solve this problem is to pre-convert the numerical representation of the input. Specifically, a large integer can be represented using a smaller number of bits by means of the reduced-radix representation. Reducing the number of bits creates space to accumulate the carries. The accumulated carries are processed at the end of the calculation, allowing the rest of the calculation to be processed in parallel with SIMD instructions.

This thesis combines the reduced-radix representation with 512-bit-wide SIMD instructions to speed up large integer multiplication and division. The target instructions are Intel Advanced Vector Extensions-512 (AVX-512) on Intel processors and Arm Scalable Vector Extension (SVE) on Arm processors. Programs implemented for these processors are executed and compared in terms of execution time with the GNU Multiple Precision Arithmetic Library (GMP), a library that performs large integer operations using scalar instructions.

For large integer multiplication with AVX-512, the basic instruction set AVX-512F (Foundation) and the integer sum-of-products instruction set AVX-512IFMA (Integer Fused Multiply-Add) are used. The basic multiplication algorithm (Basecase method) and the Karatsuba method are applied to these instructions. Xeon Phi Knights Landing (KNL) processors and Cannon Lake microarchitecture processors are used in the performance evaluation to compare the implemented program and GMP. The results show that the program with AVX-512F applied to the Basecase method improved performance by up to 2.5x on KNL and that the program with AVX-512IFMA applied to the Karatsuba method improved performance by up to 2.97x on Cannon Lake.

Arm's SIMD instruction set, SVE, was similarly applied to large integer multiplication. Since this instruction set has instructions that perform 64-bit multiplication, this study proposes an algorithm to keep the reduced-radix representation consistent. In the performance evaluation, an implementation program that combines this algorithm and the SVE instruction set is run on Fujitsu's A64FX processor. The program is compiled in two modes: one the Fujitsu compiler-based trad mode and the other the Clang/LLVM-based clang mode. The performance gains are up to 36% with trad mode compilation and up to 31% with clang mode compilation compared with GMP.

For large integer division, AVX-512IFMA instructions are used. A multiplication-based division algorithm with inverses can be applied for this task. Because AVX-512IFMA takes only 52 bits of the multiplication operand, it is necessary to properly handle the carry to ensure correct calculation. However, handling the carry is not an essential part of the calculation and should be avoided as much as possible. Therefore, this thesis proposes a division algorithm that is more suitable for SIMD instructions. In the performance evaluation, the execution time for large integer division with various operand sizes is measured on a Cannon Lake processor. The results show an average performance improvement of 25% to 35% over GMP.

These results demonstrate that SIMD instructions and the reduced-radix representation can be used to accelerate large integer multiplication and division. In the future, high-speed large integer multiplication and division can be performed using SIMD instructions on various computers.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Single instruction, multiple data (SIMD) instructions process multiple data in parallel with a single instruction. This enables parallel computing at finer granularity than that achieved using multiprocessing. Intel MMX, announced in 1996 [1], was the first widely used SIMD instruction set for general desktop computers. This instruction set has eight 64-bit registers (MM0 - MM7) and can process two 32-bit integers, four 16-bit integers, or eight 8-bit integers with a single instruction. The AltiVec extension for the PowerPC was used for media processing applications [2]. Later, Intel processors with Streaming SIMD Extensions (SSE) became available [3]. This instruction set allows SIMD operations using 128-bit-wide registers, but it processes only four 32-bit single-precision floating-point numbers. The variety of instructions was increased in SSE2 and SSE3. The SSE2 instruction set can handle two 64-bit double-precision floating-point numbers, two 64-bit integers, and four 32-bit integers. SSE evolved into Advanced Vector Extensions (AVX) [4], which includes 256-bit-wide vector operations, and then to AVX-512, which includes 512-bit-wide vector operations. SSE 4.2 implements instructions for strings (String and Text New Instructions (STTNI)). Its performance has been evaluated [5].

AVX-512 was initially used in central processing units (CPUs) for servers (e.g., Intel Xeon Phi Knights Landing architecture [6]). It is now also used in CPUs for desktop computers, such as Skylake and Ice Lake. SIMD instructions have evolved over a long period of time, becoming available in CPUs for general use. Instructions have been added to AVX-512 for cryptography [7] and deep learning [8].

The A64FX [9], an Arm processor developed by Fujitsu, and the AWS Graviton 3 can execute Scalable Vector Extension (SVE) instructions. This instruction set provides features that enable vector-length-agnostic (VLA) programming [10]. It supports SIMD instructions ranging in length from 128 bits to 2,048 bits. The A64FX and the AWS Graviton 3 are currently used in servers. Desktop computers based on the AArch64 architecture, such as those with the Apple M1 processor, are becoming more popular. Therefore, although SVE instructions are currently available for server processors such as the A64FX, it is likely that general-purpose Arm processors will eventually be able to execute these instructions.

SIMD instructions are suitable for computer arithmetic, such as multiplication and division. In a multi-precision calculation, the same operation is often performed on multiple data due to the large number of elements. SIMD instructions are a good match for this task. Since multiplication and division are more complex and time-consuming than addition and subtraction, it is important to optimize them. Large integer arithmetic, which cannot be handled by a single instruction in a typical processor, is used in cryptography, such as Rivest-Shamir-Adleman (RSA) cryptography [11] and elliptic curve cryptography [12], and computer algebra systems, such as Maxima [13] and Wolfram Mathematica [14]. There are libraries, such as the GNU Multiple Precision Arithmetic Library (GMP) [15], for high-speed arbitrary-precision arithmetic. GMP is implemented with scalar instructions for arithmetic calculations. Mathematica performs multi-precision operations using GMP [16]. SIMD instructions can process more data than can scalar instructions and thus have the potential to be faster for large integer multiplication and division.

A graphics processing unit (GPU) can be used to process large integer arithmetic operations at high speed. GPUs are used not only for graphics applications such as ray tracing, but also for large-scale computing applications, especially for machine learning. Several studies have performed large integer calculations on GPUs [17] [18]. Since GPUs have a very large number of cores, they are especially suitable for large computations. However, unlike CPUs, GPUs are not available in every computer. Furthermore, the range of operand sizes that can be processed at high speed differs between CPUs and GPUs. Running large integer arithmetic operations on the CPU thus allows large computations to be executed on any computer. As mentioned above, it is expected that desktop CPUs with wide SIMD instructions such as AVX-512 and SVE will become increasingly common. This thesis demonstrates that large integer multiplication and division can be computed at high speed on any computer using SIMD instructions.

1.2 Objective

The objective of this thesis is to improve the performance of large integer multiplication and division by using SIMD instructions on CPUs.

Multiplication using AVX-512 and SVE is first implemented and evaluated. There are several multiplication instructions included in these instruction sets, each with different properties. For example, AVX-512IFMA takes only 52 bits of the multiplication operand and SVE can retrieve the upper word of a 64-bit multiplication. Therefore, in this thesis, implementation methods and algorithms tailored to the characteristics of each instruction are proposed.

Next, division using AVX-512 is implemented and evaluated. For large integer division, the effect of AVX-512IFMA on execution time is evaluated for a method for computing large integer division that uses multiplication instead of division instructions. In addition, since the division algorithm is more complex than the multiplication algorithm, implementation methods and algorithms for making vectorization more efficient using SIMD instructions are proposed.

For both multiplication and division, the calculations are performed using the reduced-radix representation. The speedup effect of this representation on multiplication and division is evaluated. The implementations are run on the target processors and their performance is compared with that of GMP in terms of execution time. This thesis mainly focuses on naive algorithms and assumes that the implementations are run in a single thread on a single core. Therefore, the main target operand size will be thousands to tens of thousands of bits. In this range, performing calculations with multi-core and multi-thread has a large overhead associated with it. However, once SIMD instructions speed up the essential part of the computation, the concept can be applied to computation in range where parallelization is required. As another application, in situations where multiple large integers are processed in parallel, such as in matrix calculations, the per-core speedup also contributes to overall speedup. These evaluations confirm that SIMD instructions can speed up large integer multiplication and division.

1.3 Contributions

The main contributions of this thesis are as follows.

First, it is shown that Intel AVX-512 instructions, in particular AVX-512IFMA, speed up large integer multiplication.

Second, it is shown that Arm SVE instructions also speed up large integer multiplication. An algorithm that uses the reduced-radix representation is proposed for ordinary multiplication instructions (i.e., not special instructions such as AVX-512IFMA). This algorithm is shown to be faster than GMP. This also suggests that Intel processors are not the only ones capable of speeding up large integer arithmetic operations.

Third, it is shown that AVX-512 can also speed up large integer division. An algorithm that is better suited to SIMD instructions than are conventional division algorithms is proposed. This algorithm is shown to be faster than GMP's division function.

Finally, it is shown that the reduced-radix representation facilitates the use of SIMD instructions for computing large integer multiplication and division. Although the conversion to this representation requires additional processing, it is shown that the resulting arithmetic operations are faster than those of GMP because they take full advantage of SIMD instructions.

In summary, it is shown that SIMD instructions speed up the multiplication and division of large integers. For both multiplication and division, the advantage of the implementation based on SIMD instructions increases with increasing operand size. Therefore, algorithms based on these instructions should outperform arbitrary-precision arithmetic libraries such as GMP. Large integer multiplication and division using SIMD instructions can be processed at high speed on a lot of computers, which could speed up various processes.

1.4 Organization of Thesis

The rest of this thesis is organized as follows. Chapter 2 describes the reduced-radix representation and SIMD instructions. Chapter 3 describes large integer multiplication using AVX-512 instructions. Two instruction sets are covered here, namely AVX-512F and AVX-512IFMA. Chapter 4 describes large integer multiplication using SVE instructions. Chapter 5 describes large integer division using AVX-512 instructions. Finally, Chapter 6 concludes this thesis and discusses future work.

Chapter 2

Preliminaries

2.1 Reduced-Radix Representation

First, the representation of large integers used in this study is discussed. An n-word integer A is expressed as follows:

$$A = \sum_{i=0}^{n-1} a_i \beta^i,$$

where a_i is a word value and β depends on the bit width of one word (e.g., 2^{64} in arrays of 64-bit integers). Since this study is concerned with unsigned integers, large integers are stored in unsigned integer arrays in the implementation. In this thesis, large integers are divided into 32- or 64-bit/word units for multiplication and division. The 64-bit/word unit case is discussed in this section.

For large integer multiplication, the partial products obtained by multiplying individual words are added. A 128-bit partial product is obtained from word-by-word multiplication. For example, for an x86 Intel processor with scalar instructions, the partial product obtained from the `MUL` instruction is divided into a high part and a low part and stored in two 64-bit registers [19]. The registers that contain the appropriate partial products are then added. Since each register has 64 bits, the result of the addition may be 65 bits. This means that if a carry occurs, the value will not fit in the register and thus an erroneous calculation result would be obtained. This is not much of a problem if the calculation is being done with scalar instructions because another status flag is used to hold the carry (i.e., the carry is not lost). For Intel processors, the EFLAGS register holds the carry flag (CF) [19]. For Arm processors, the Processor State (PSTATE) holds the carry flag (C) [20]. When adjacent partial products are added, an `ADC` instruction, which performs the addition of the two registers and the carry condition flag to get the correct calculation result, is executed. However, SIMD instructions such as AVX-512 and SVE do not have registers that hold such carries. They do not have an `ADC`-like instruction and thus carries generated by the normal `ADD` instruction disappear [19] [21]. Therefore, it is necessary to carefully consider carry propagation when using SIMD instructions.

One way to solve these problems is to not perform multiplication on all 64 bits. That is, each operand is converted into a representation with fewer than 64 bits prior to multiplication. In this approach, a word is represented as a value using, for example, 56 or 52 bits and the remaining upper bits are set to zero. With this representation, called the reduced-radix representation [22], a carry does not exceed the width of the register (it is stored in the upper bits). The upper bits are used to store the carries generated during the addition of partial products. Thus, if partial products are added using the reduced-radix representation with the same radix, the sum can be computed without losing carries (until there is no more space to hold them). Furthermore, since borrows can be stored using the same process, subtraction can be performed as well.

This representation increases computational efficiency. In general, when addition with a carry is performed on a word, it is difficult to vectorize the operation because there is a data dependency between a word and the subsequent word. In [23], methods for detecting overflow without the EFLAGS register are described. If these methods are applied to SIMD instructions, overflow can be detected to prevent bit loss. However, since these detection methods are carried out for every addition, the cost increases as the number of words increases. For n-word multiplication, the computational complexity of this process is $O(n^2)$ and thus the total cost is very large for large n. This representation, however, allows us to conduct parallel processing using SIMD instructions because the carries are not lost. The accumulated carries are processed only when there is no more space to hold carries or when all partial products have been added. The computational complexity for carries is $O(n)$, which is lower than that of the above process. For the former case, the carries are processed because any further addition will result in their loss. However, this processing may affect the execution time for large integer multiplication operations and should be avoided as much as possible. Therefore, it is better to make the radix fit the target in advance, especially if the scale of the target operation is so small that the execution time is very short.

The feasibility of using the reduced-radix representation is now discussed. If an 8-bit space can be created by reducing each word from 64 bits to 56 bits, 255 ($= 2^8 - 1$) times 56 bits + 56 bits can be executed to accumulate carries. Generally, when two n-word multi-precision integers are multiplied, $n - 1$ additions are performed at most. Therefore, it is theoretically possible to perform multiplication without carry propagation up to a maximum of 256 ($= 2^8$) words (i.e., 56 bits $\times$ 2^8 $=$ 14,336 bits). With a similar assumption for a 52-bit representation, a multiplication of 212,992 ($= 52 \times 2^{64\text{-}52}$) bits in the 2^{52}-radix representation may exceed the capacity of the remaining 12 bits as a result of adding partial products. However, this size is so large that the computation time for the multiplication itself is also long. Therefore, for a multiplication that exceeds this size, it is reasonable to assume that the execution time to clear the 12-bit carries during the calculation of the multiplication is sufficiently small relative to the overall execution time.

In summary, the reduced-radix representation prevents carry loss and enables parallelization via SIMD instructions. This representation is thus used for multiplication and division operations in this thesis.

2.2 SIMD Instructions

2.2.1 Intel AVX-512

Intel AVX-512 is a set of 512-bit-wide SIMD instructions [24] that can handle eight 64-bit double-precision and 16 32-bit single-precision floating point instructions, as well as eight 64-bit and 16 32-bit integers. AVX-512 can thus deal with more data at a given time than can 128-bit SSE instructions or 256-bit AVX instructions. Furthermore, AVX-512 has more containers (32 zmm registers) than do SSE and AVX, and thus AVX-512 has the potential to reduce the replacement of register data and memory data due to register insufficiency (i.e., register spilling). In addition, eight mask registers (k0 - k7) are available in AVX-512, allowing selective processing of the data in the registers. It is possible to vectorize the processing of loops that contain conditional statements, which is difficult to achieve with SIMD instructions. Knights Corner, the processor generation before Knights Landing, includes a set of 512-bit vector instructions called Intel IMCI (Initial Many Core Instructions) [25]. Although Intel IMCI is similar to AVX-512, it is not directly compatible.

AVX-512 has evolved to be able to execute many types of instruction. For example, AVX-512F (Foundation) was initially offered as a basic instruction set, but later it became capable of executing more granular instructions such as DQ (Doubleword and Quadword), BW (Byte and Word), and VBMI (Vector Byte Manipulation Instructions), and also of performing multiply-and-accumulate operations such as IFMA (Integer Fused Multiply-Add). Recently, domain-specific instructions such as GFNI (Galois Field New Instructions) have also been included [26].

For AVX-512 instructions, the code presented here uses the Intel intrinsic instructions, which are C style functions that provide access to AVX-512, AVX, SSE, and MMX [27]. These built-in functions, instead of assembly code, can be used to manipulate SIMD instructions, which increases the readability and availability of source code. Intrinsic functions, which can be compiled using GCC (the GNU compiler collection) or the Intel compiler, prevent register misassignment due to the manual writing of assembly code (i.e., the compiler assigns all registers). In particular, the Intel compiler considers dependencies and optimizes instructions for a processor to execute instructions collectively in one cycle. In summary, the intrinsic functions allow SIMD instructions to be manipulated faster, more accurately, and more effectively than can be achieved by writing lower-level code.

2.2.2 Arm SVE

Arm SVE is an instruction set for the Arm AArch64 architecture [28]. In this instruction set, 32 scalable vector registers (Z registers) and 16 predicate registers (P registers) are available. These registers are similar to the zmm and mask registers in AVX-512, respectively. The critical difference is that the vector length of the AVX-512 instruction set is fixed at 512 bits, whereas SVE scalable registers can be from 128 to 2,048 bits wide. This flexibility allows bit-width-independent code to be written and executed on another SVE-supported processor without recompilation. The A64FX processor, the target processor in this thesis, is capable of executing 512-bit-wide SIMD instructions. SVE has the basic instructions available. The scope of this thesis covers the processing that can be done with AVX-512, with the exception of AVX-512IFMA.

SVE instructions can be implemented using intrinsic functions similar to those for AVX-512 [21]. These functions can be treated in the same way as C functions to use the desired SIMD instructions. However, because SVE intrinsic functions are well-defined, it is necessary to specify the data type more strictly than is done for AVX-512. Furthermore, with the exception of some instructions, the predicate registers must be passed explicitly even if they do not need to be masked.

Chapter 3

Large Integer Multiplication Using Intel AVX-512

This chapter focuses on large integer multiplication using Intel AVX-512. AVX-512 operations are applied to the reduced-radix representation and the implementation is evaluated on an Intel Xeon Phi processor and the Cannon Lake microarchitecture. Although the number of bits is limited, a kernel that can multiply variable-length operands rather than specialized operands is implemented. This chapter describes two types of implementation, one using AVX-512F and the other using AVX-512IFMA.

3.1 Related Works

Several studies have attempted to speed up large integer arithmetic processing using Intel's SIMD instructions.

A program that applies SSE2 to the reduced-radix representation was compared with a naive implementation based on simple scalar operations on an Intel Pentium 4 processor [22]. A speedup of approximately 10.7x was achieved by using SSE2. Gueron and Krasnov implemented a multiplication program with AVX2 instructions and the reduced-radix representation for modular arithmetic [29]. They patched the program into OpenSSL [30] and evaluated the processing of RSA cryptography with the Intel Software Developer Emulator (SDE). The program achieved a reduction of approximately 50% in both the number of instructions and the number of cycles compared with the original OpenSSL. Moreover, AVX instruction sets have been used for modular arithmetic in the context of polynomials. AVX2, a 256-bit-wide SIMD instruction set, and AVX-512 have yielded performance improvements of about 5x and 10x, respectively, over scalar instructions [31]. Keliris and Maniatakos implemented large integer multiplication with AVX-512F instructions using a 2^{29}-radix representation to avoid carry propagation as much as possible [32]. When the number of words exceeded a certain threshold, carries were cleaned up. They also evaluated the kernel on the SDE in terms of the number of instructions. In their paper, the SDE was used to emulate computation on a Knights Landing processor. Although cleanup processing was carried out, a 1.16x improvement in performance was still recorded compared with GMP in the context of 2,048-bit multiplication.

Several studies have evaluated AVX-512IFMA as well. Gueron and Krasnov evaluated multiplication using AVX-512IFMA [33]. In their research, as well as in [32], the number of bits per word was reduced to prevent carry propagation. Since no processor was available that could execute AVX-512IFMA at the time their study was conducted, the implementation was evaluated using the Intel SDE in terms of instruction count. Large integer multiplication with four sizes, namely 1,024, 2,048, 3,072, and 4,096 bits, was evaluated. The performance evaluation focused on implementations that used the conventional AVX-512F, GMP, and AVX-512IFMA. Despite the conversion to the reduced-radix representation and the conversion back to a normal representation, the number of instructions for 4,096-bit multiplication was about one-eighth that for GMP and about one-quarter that for AVX-512F. Looking at another aspect, AVX-512IFMA has also been used to speed up modular squaring [34]. A study implemented squaring with Montgomery multiplication [35] (referred to as Almost Montgomery Squaring in [34]). Further evaluation in terms of latency, throughput, and number of instructions revealed that the implementation using AVX-512IFMA instructions was the most effective. The authors concluded that Almost Montgomery Squaring is faster than the conventional calculation of modular exponentiation and that AVX-512IFMA instructions have the potential to speed up the calculation of modular exponentiation.

In other related research, SIMD instructions, such as SSE, AVX, and AVX-512, were applied to large integer arithmetic, with excellent results. However, no research has measured the execution time for large integer multiplication with AVX-512 and the reduced-radix representation on a real processor. In [32] and [33], which are the most relevant to the present study, the authors used the SDE and measured only the number of instructions because no processor was available at the time. Since Intel Xeon processors that can execute AVX-512 instructions decrease their operating frequency by several hundred megahertz at high AVX frequencies [36], the time taken for one cycle is different compared with that for scalar instructions. Thus, a program with AVX-512 instructions cannot be simply compared with a program without AVX instructions in terms of the number of instructions. In this chapter, the execution time is measured.

3.2 Multiplication Using AVX-512F

This section describes large integer multiplication based on AVX-512F, which contains basic operations. For AVX-512F, the multiplier and multiplicand specified as the argument of the kernel are arrays of 32-bit unsigned integers.

3.2.1 Basecase Multiplication

The Basecase multiplication algorithm [37], which is the most basic multiplication algorithm, is used here. The Basecase multiplication algorithm is implemented using vector operations, making it different from algorithms that use scalar operations.

Algorithm 1 Basecase multiplication in radix 2^N based on vector operations

```
Input: $A = \sum_{k=0}^{m-1} a_k\beta^k, B = \sum_{k=0}^{n-1} b_k\beta^k$
Output: $C = A \times B := \sum_{k=0}^{m+n-1} c_k\beta^k$
 1: $i \leftarrow 0$
 2: while $i < n$ do
 3:     $j \leftarrow 0$
 4:     while $j < m$ do
 5:         $\boldsymbol{a} \leftarrow [a_j, a_{j+1}, a_{j+2}, \ldots, a_{j+(\mathrm{VLEN}-1)}]$
 6:         for $k = 0$ to VLEN$-1$ do
 7:             $\boldsymbol{c} \leftarrow [c_{j+k}, c_{j+k+1}, c_{j+k+2}, \ldots, c_{j+k+(\mathrm{VLEN}-1)}]$
 8:             $\boldsymbol{c} \leftarrow \boldsymbol{c} + \boldsymbol{a} \times b_{i+k}$
 9:         end for
10:         $j \leftarrow j + \mathrm{VLEN}$
11:     end while
12:     $i \leftarrow i + \mathrm{VLEN}$
13: end while
14: convert $C$ back to $2^N$-radix representation
```

Algorithm 1 is the Basecase multiplication kernel with vector operations. Note that the multiplier and multiplicand are radix $\beta = 2^{28}$ ($N = 28$) for AVX-512F here. In addition, since AVX-512 is used, VLEN $= 8$. In the implemented program, $\boldsymbol{c}$ is stored in an appropriate position in memory after line 8. For the next round, the kernel loads partial products shifted by one word to $\boldsymbol{c}$ and repeats the processing loop. AVX-512 instructions are used on lines 5, 7, and 8 since these lines contain vector calculations.

In the present study, since the reduced-radix representation is used, it is assumed that both the multiplier and multiplicand are within 7,168 bits and thus the kernel runs the loop without carry processing. After the loops, C is a radix 2^{64} number due to 32-bit multiplication and thus C must be converted back into a radix 2^N number.

Algorithm 2 Conversion of α-radix number into β-radix number

```
Input: $X = \sum_{k=0}^{m+n-2} x_k \alpha^k$
Output: $C = \sum_{k=0}^{m+n-1} c_k \beta^k$
1: for $i = 0$ to $m + n - 3$ do
2:     $c_i \leftarrow x_i$ mod $\beta$
3:     $d \leftarrow x_i / \beta$
4:     $x_{i+1} \leftarrow x_{i+1} + d$
5: end for
6: $c_{m+n-2} \leftarrow x_{m+n-2}$ mod $\beta$
7: $c_{m+n-1} \leftarrow x_{m+n-2} / \beta$
```

Algorithm 2 describes the process of converting an α-radix number into a β-radix number. In this paper, $\alpha = 2^{64}$, $\beta = 2^{28}$, and d is the carry. Although there is a modular operation on line 2, c_i can be readily calculated by an AND operation with $\beta - 1$ because β is a power of two. Concretely, the AND operation is applied to x_i and 0xfffffff $(= 2^{28} - 1)$. Similarly, the division on line 3 can be dealt with by a shift operation, namely a 28-bit right shift. After loop processing, only the last word of X remains. Then, the results of the modular and division operations are stored in c_{m+n-2} and c_{m+n-1}, respectively. This process is sequential when scalar instructions are used because of the data dependency between a word and its neighboring words.

3.2.2 Implementation

AVX-512 Instructions for Implementation

The major AVX-512F intrinsic functions and their corresponding instructions used in the proposed kernel are given below. Here, `epi` and `epu` denote extended packed integer and extended packed unsigned integer, respectively, and the last number indicates the number of bits.

`_mm512_mul_epu32`
: Multiply the lower 32 bits of an unsigned 64-bit integer and store the result as an unsigned 64-bit integer in the destination register. This function is converted into `vpmuludq`.

`_mm512_add_epi64`
: Add 64-bit integers. Its instruction is `vpaddq`. If a carry occurs through this instruction, the overflowed bit is lost.

`_mm512_permutexvar_epi32`
: Shuffle the contents of the source register according to the index register. This function is paired with `vpermd`. This instruction shuffles 32-bit integers and generates the b_{i+k}-vector on line 8 of Algorithm 1.

When an intrinsic function of an AVX-512 instruction is used with the C language, `__m512` type variables are declared. When these variables are passed to the function, the registers that contain its value are specified for the corresponding AVX-512 instruction. To execute the AVX-512 instruction, no special C code is needed for reading from memory to the register or writing from the register to memory. The C compiler manages all of the instructions. Therefore, SIMD operations such as AVX-512 can be handled without worrying about registers. In this thesis, variables of type `__m512i` are declared because integers are used.

Figure 3.1 describes the `_mm512_permutexvar_epi32` (`vpermd`) process used to generate a b_{i+7} vector as an example. Actually, the zeros in the intermediate container are also the shuffling indices and thus b_i is stored in the white parts of the destination register in Figure 3.1. However, they have no effect on processing in the kernel because `vpmuludq` does not use the white parts. This instruction is used to multiply each element

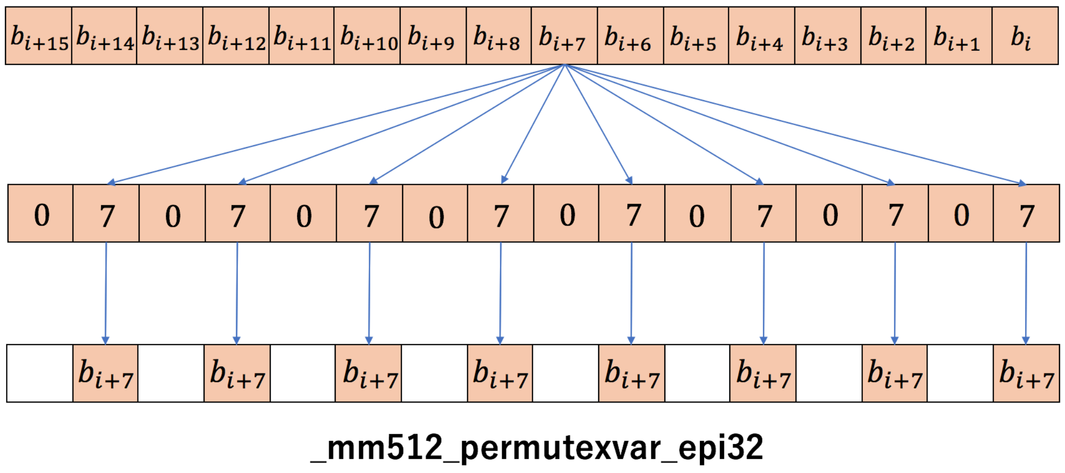

b_{i+15} b_{i+14} b_{i+13} b_{i+12} b_{i+11} b_{i+10} b_{i+9} b_{i+8} b_{i+7} b_{i+6} b_{i+5} b_{i+4} b_{i+3} b_{i+2} b_{i+1} b_i
0 7 0 7 0 7 0 7 0 7 0 7 0 7 0 7
b_{i+7} b_{i+7} b_{i+7} b_{i+7} b_{i+7} b_{i+7} b_{i+7} b_{i+7}
_mm512_permutexvar_epi32

Figure 3.1: Overview of `_mm512_permutexvar_epi32` processing.

of the multiplicand vector by one element of the multiplier. If one element is simply copied to the entire zmm register, this process can be accomplished by reading an element from the array of multipliers and broadcasting the element. AVX-512F provides an equivalent function, namely `_mm512_set1_epi32` (`vpbroadcastd`). However, memory reads for each loop of Algorithm 1 would result in high latency. Therefore, data are loaded for the zmm register width before the innermost loop is entered. Then, `vpermd` retrieves the necessary data from the register. This method reduces the number of memory accesses.

Figure 3.2 describes the computation on line 8 in Algorithm 1 through three intrinsic functions, as mentioned above. In this figure, c' terms are the partial products before multiplication and c terms are the calculation results. As shown, the calculation of the eight partial products is computed in three instructions.

Fixed-Length Multiplication Module

In this paper, a multiplication program for flexible-length operands is implemented. In the multiplication of multi-precision integers, the kernel itself does not know the number of argument words in advance because the kernel is not implemented for fixed-length operands, and thus the kernel must be able to handle variable-length multipliers and multiplicands. To achieve this with a naive implementation, the kernel runs two `for` loops that correspond to the length of each operand. However, the numbers of branches and arithmetic operations increase with the square of the number of words in the operands. AVX-512 instructions can

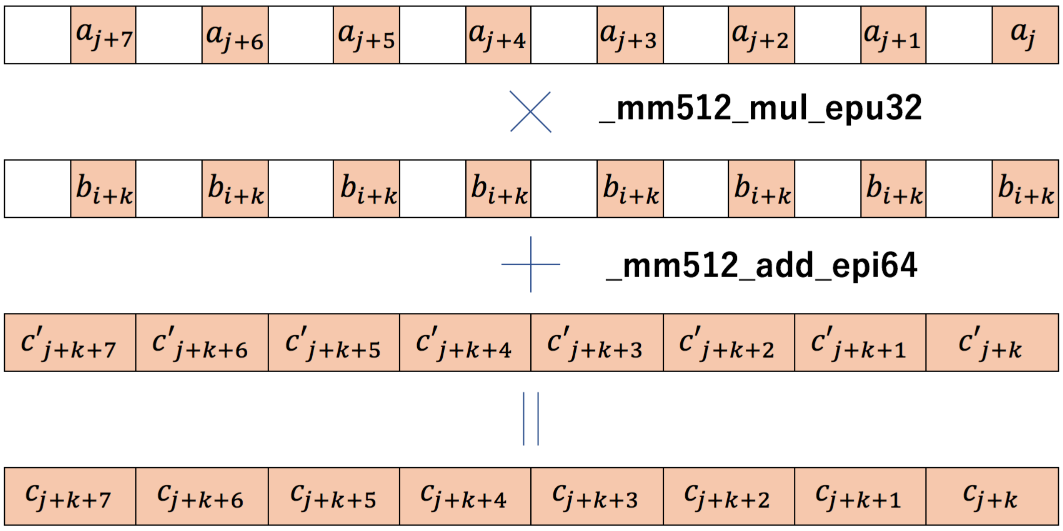


Figure 3.2: Process on line 8 in Algorithm 1.

handle 224 bits ($=$ 28 bits $\times$ 8 words) simultaneously with a 2^{28}-radix representation, and hence an n-bit multiplication has $\lceil n/224 \rceil^2$ branches. As an example, Figure 3.3 describes the combination of multiplied elements in 2,048-bit multiplication with a simple implementation. The top and bottom sets of elements (i.e., a_{0-73} and b_{0-73}, respectively) indicate the 2,048-bit numbers. There are 100 pairs of elements, and thus the kernel executes 100 loops in this example. The overhead due to forks should be minimized for optimization. If the number of branches involved in the loop can be reduced, the overhead will also be reduced.

To handle this issue for flexible-length operands, a small-scale module that multiplies fixed-length operands is implemented and called repeatedly. Since this module is invoked in the interface function that users actually call, users are not conscious of the module. Its argument size is locked so that the number of multiplications in this module is also locked. It is possible to unroll the loop and optimize the implementation within it. In this section, the implemented module collectively computes 32 words of each operand. For example, the module deals with 896 bits ($=$ 28 bits $\times$ 32 words) at once. This module originally runs for 16 loops, but they are all unrolled. Consequently, there are no branches in the module. n-bit multiplication processes $\lceil n/896 \rceil^2$ loops. Figure 3.4 displays a diagram of the element combinations in 2,048-bit multiplication using the 896-bit fixed-length module as an example. In this multiplication, the module is called nine times in total. Therefore, this figure indicates that the module has 91 fewer loops compared with the example in

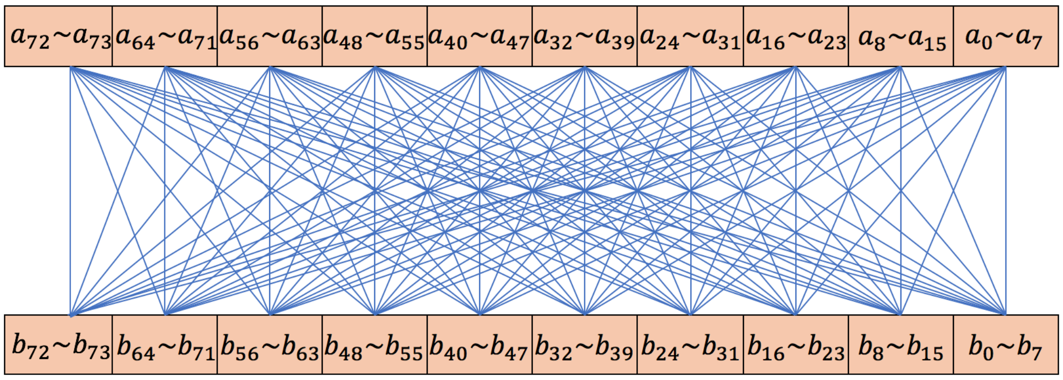


Figure 3.3: Naive 2,048-bit multiplication. It is necessary to loop through 100 pairs.

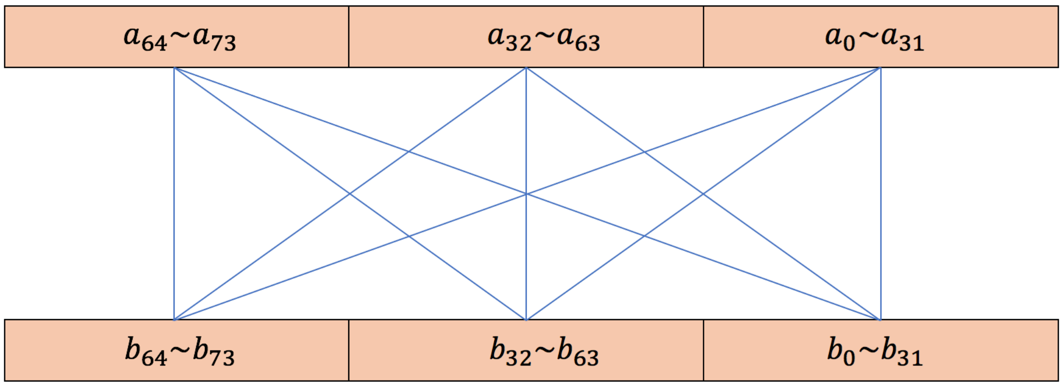


Figure 3.4: 896-bit fixed-length module used in 2,048-bit multiplication. Since there are only nine pairs here, the number of loops is reduced compared with that in Figure 3.3.

Figure 3.3 and that it can deal with flexible-length multiplication.

Distribution of Partial Products

The internal implementation of the module explained in the previous section is discussed here. The kernel calculates the partial products in loops in Algorithm 1 while shifting the partial product by one word. However, there is a data dependency between the $\boldsymbol{c}$ on line 8 and the $\boldsymbol{c}$ on line 7 of the next `for` iteration. Hence, until the process on line 8 is completed, the program cannot execute the next `for` iteration. Subsequently, a stall occurs, preventing pipeline parallelism. According to [22], two arrays (t and u) for partial products are prepared. t and u contain the partial products from even and odd words, respectively, of a multiplier B and are independent of each other. Concretely, the kernel can avoid a stall

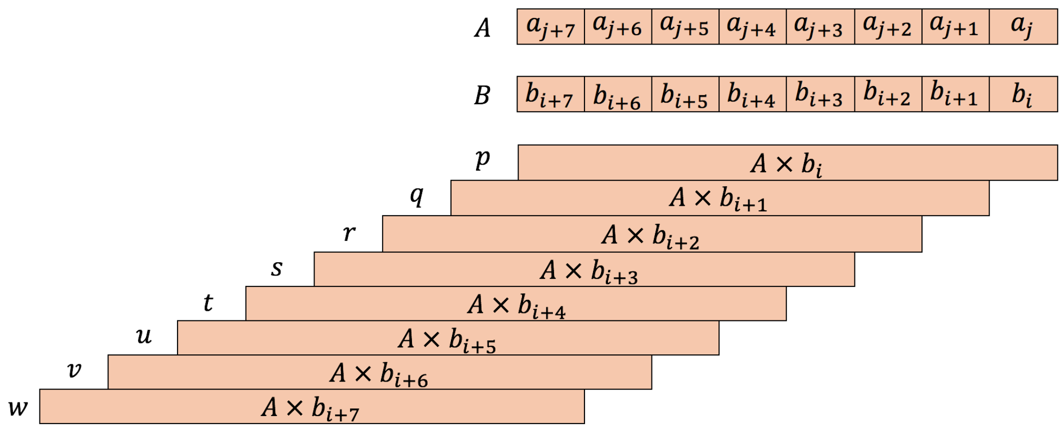


Figure 3.5: Use of arrays of partial products.

because it stores data in t on line 8 and loads data from u on line 7 of the next `for` iteration. The same is true even if t and u are exchanged. Therefore, the kernel can be executed in parallel through pipeline processing.

Figure 3.5 summarizes multiplication with partial product arrays. Since VLEN $= 8$, $p, q, \ldots, w$ arrays are prepared and $A \times b_{i+k}$ $(0 \leq k \leq 7)$ is stored in the array. When all processing is complete, the values of those arrays are added at the appropriate locations. Finally, the function returns the multiplication result.

A naive implementation is as follows.

1. Load operand data from memory into $\boldsymbol{p}$.

2. Calculate $\boldsymbol{p} + \boldsymbol{a} \times b_i$.

3. Store $\boldsymbol{p}$ in memory.

Partial products $q, r, \ldots$, and w are processed in the same way. However, there are data dependencies between steps 1 and 2 and between steps 2 and 3 and hence stalls occur between these steps. For any set of arrays, the naive implementation prevents pipeline parallel processing.

To overcome this problem, a design that collectively deals with multiple partial products instead of single arrays one by one is adopted. Figure 3.6 shows pseudo code that handles multiple arrays as a group. Eight arrays are divided into four groups: $\{p, q\}$, $\{r, s\}$, $\{t, u\}$, and $\{v, w\}$. These independent containers can be arranged such that dependency-free processes can be inserted between the loading, product-sum, and storage stages. Specifically,

```
1: Load $\boldsymbol{p}$
2: Load $\boldsymbol{q}$
3: $\boldsymbol{p} \leftarrow \boldsymbol{p} + \boldsymbol{a} \times b_i$
4: $\boldsymbol{q} \leftarrow \boldsymbol{q} + \boldsymbol{a} \times b_{i+1}$
5: Store $\boldsymbol{p}$
6: Store $\boldsymbol{q}$
7: Process for $\{r, s\}$, $\{t, u\}$, $\{v, w\}$ in the same way
```

Figure 3.6: Pseudo code with arrays of partial products.

the codes for $\boldsymbol{q}$ that have no dependency on $\boldsymbol{p}$ are inserted between lines 1 and 3 and between lines 3 and 5 in this code. This avoids stalls and conceals latency compared with the naive implementation.

The eight arrays can be divided in other ways, such as into eight sets (one by one), two sets, and one set. That is, ($\{p\}$, $\{q\}$, $\{r\}$, $\{s\}$, $\{t\}$, $\{u\}$, $\{v\}$, $\{w\}$), ($\{p, q, r, s\}$, $\{t, u, v, w\}$), and ($\{p, q, r, s, t, u, v, w\}$), respectively. As mentioned above, if eight arrays are divided into eight sets (i.e., naive implementation), then pipeline parallel processing of the kernel cannot be performed. The other methods also increase stall. For example, when four variables are handled at once, three independent code sections can be inserted between lines 1 and 3 in Figure 3.6. Similarly, when eight variables are handled at once, seven independent code sections can be inserted. Although these methods increase pipeline parallelism compared with that obtained with four groups, stalls occur because there are more multiple memory accesses. In particular, in the middle stage, several load instructions are performed immediately after multiple store instructions. For example, when eight sets are used, the kernel processes eight stores and eight loads and performs a memory access 16 times. As a result, the number of stalls increases. Based on the output from the Intel compiler assembler for each grouping, the smallest number of stalls is achieved when the eight elements are grouped into four sets. Therefore, it can be concluded that the eight partial products are best processed in four groups.

3.2.3 Evaluation

Evaluation Environment

To evaluate performance, a host computer with an Intel Xeon Phi 7250 (Knights Landing) CPU (1.40 GHz) and 16 GB MCDRAM + 96 GB DDR4 memory running the CentOS 7.3 operating system was used. In the evaluation, unless otherwise noted, MCDRAM through numactl [38] was used. The cluster mode of Knights Landing is quadrant mode. The experimental programs were implemented in the C language and compiled using the Intel C compiler `icc` version 17.0.1 with the `-O3 -xMIC-AVX512 -fno-alias -std=c99` options. SDE version 8.12.0 was used to count instructions. The comparison target was GMP version 6.1.2, which was built with `icc`. All programs were executed on a single core with a single thread. The multiplier and multiplicand, which are arguments of the large integer multiply function, were generated by the rand function. The system time was used as the random seed.

Instruction Count Comparison

The instruction count for the function designed in the previous sections was measured. Table 3.1 shows the instruction count for GMP 6.1.2, the program in [33], the program in [32], and the proposed program. The exact numbers of instructions were not reported in [33] and thus the values for this program were estimated from the reported graphs. In the table, *Ratio* is the ratio of the number of instructions for the given program to that for the proposed program, and *KNL* denotes Knights Landing. The values for the proposed program tend to be proportional to the number of operand bits. Compared with [33], there are large differences for all operand sizes. The program in [33] is optimized for 1,024-, 2,048-, 3,072-, and 4,096-bit operands. In contrast, the proposed program handles each operand size as flexibly as possible. Therefore, the observed differences are mainly attributed to differences in flexibility. Focusing on GMP and KNL, for the 512- and 1,024-bit operations, the number of instructions for the proposed program is lower than that for GMP and higher than that for KNL [32]. Despite the small operand, the instruction count for the proposed program is large because the number of instructions for converting back to a normal representation (i.e., the last line in Algorithm 1) is high. For the 2,048- and 4,096-bit operations, the proposed program uses the fewest instructions. For KNL, the carry processing cost increases with increasing number of words and thus the instruction count also increases.

Table 3.1: Number of instructions for proposed implementation, GMP 6.1.2, [33], and [32]

Size (bit)	Our	KNL [32]	Ratio	GMP	Ratio	AVX-512F [33]	Ratio
512	2093	1405	0.67	7334	3.50	-	-
1024	4023	3493	0.87	10886	2.71	750	0.19
2048	7985	11055	1.38	24867	3.11	2200	0.28
3072	12712	-	-	46971	3.70	4200	0.33
4096	18292	43657	2.39	62405	3.41	7800	0.43

Table 3.2: Number of instructions and execution time for GMP 6.1.2 and GMP 5.1.3 in [32]

Size (bit)	GMP 6.1.2 [inst]	GMP 5.1.3 [inst]	GMP 6.1.2 [us]	GMP 5.1.3 [us]
512	7334	4652	1.218	1.227
1024	10886	6452	3.024	3.115
2048	24867	12841	8.825	9.267
4096	62405	30247	26.619	32.325

Furthermore, the cleanup function in [32] affected this result. For GMP, the number of instructions tends to be large because scalar operations are used.

The number of instructions for GMP 6.1.2 becomes enormous as the operand size increases. The values obtained in this experiment deviate from those for GMP 5.1.3 in [32]. Table 3.2 compares their instruction counts and execution times. The number of instructions for GMP 6.1.2 is about twice that for GMP 5.1.3. In GMP 6.1.2, `__gmpn_addmul_1` is called many times inside the `mpz_mul` function, which performs multiple-precision integer multiplication. On the other hand, `__gmpn_addmul_1` is either not called or called only a few times in GMP 5.1.3. Although the instruction count in version 6.1.2 is large, a comparison of execution times shows that this version is slightly faster than version 5.1.3. From these results, it can be concluded that a small number of instructions does not necessarily lead to a short execution time and that a comparison in terms of the number of instructions alone is insufficient.

Execution Time Comparison

Unfortunately, the *AVX-512F* program in [33] uses AVX-512BW and AVX-512VL instructions and thus it cannot be executed on the Knights Landing processor. In this section, the execution times of the proposed implementation and GMP are compared. Figures 3.7 and 3.8 show graphs of the execution time for the proposed implementation and GMP. The pro-

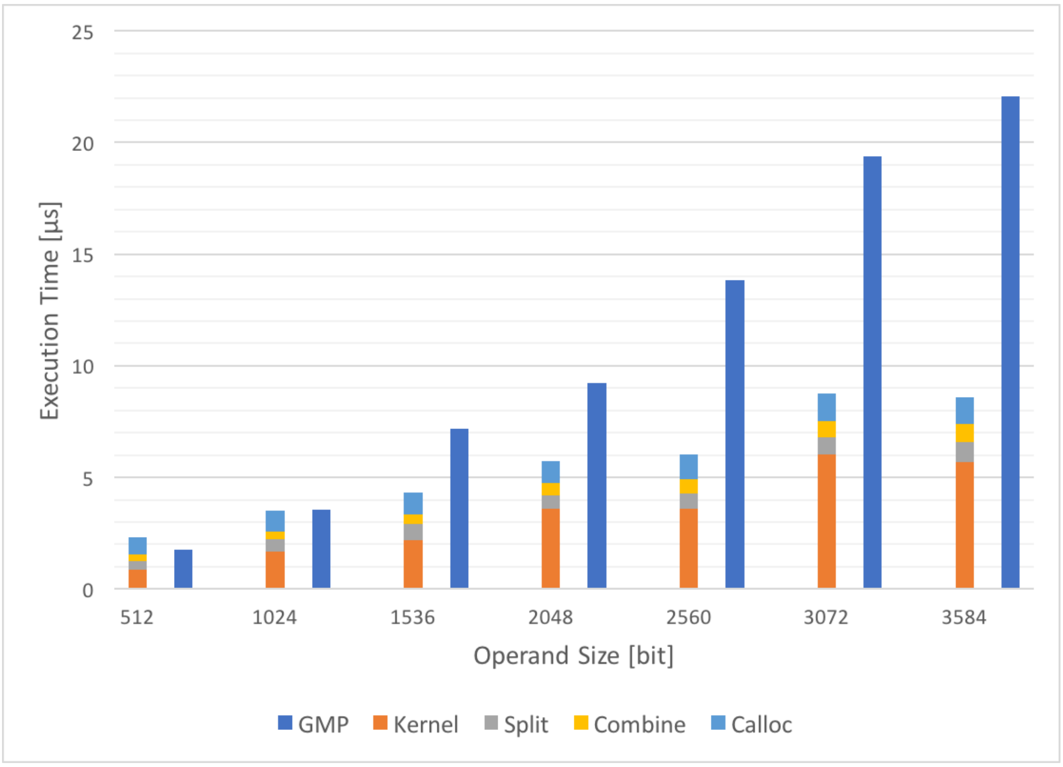


Figure 3.7: Execution time of multiplication function in proposed program and GMP 6.1.2 for 512- to 3,584-bit operands.

gram time for the proposed implementation consists of Kernel, Calloc, Split, and Combine. Split is the conversion to the reduced-radix representation, and Combine is the reverse process. Although the proposed program is inferior to GMP in terms of performance at 512 bits, it is superior to GMP at 1,536 bits or more. These programs have approximately the same execution time at 1,024 bits. The maximum performance improvement rate is approximately 2.5x. The result at small operand sizes can be explained by the time of `calloc` for partial products and the conversion function accounting for most of the total execution time. For operand sizes of 512 bits to 1,536 bits, the time for conversion and allocation account for approximately half of the total execution time.

There is almost no difference in processing time between 1,024- and 1,536-bit operands, 2,048- and 2,560-bit operands, 3,072- and 3,584-bit operands, 4,608- and 5,120-bit operands, 5,632- and 6,144-bit operands, and 6,656- and 7,168-bit operands. This is due to the fixed-length multiplication module described in Section 3.2.2. With radix 2^{28}, this module calcu-

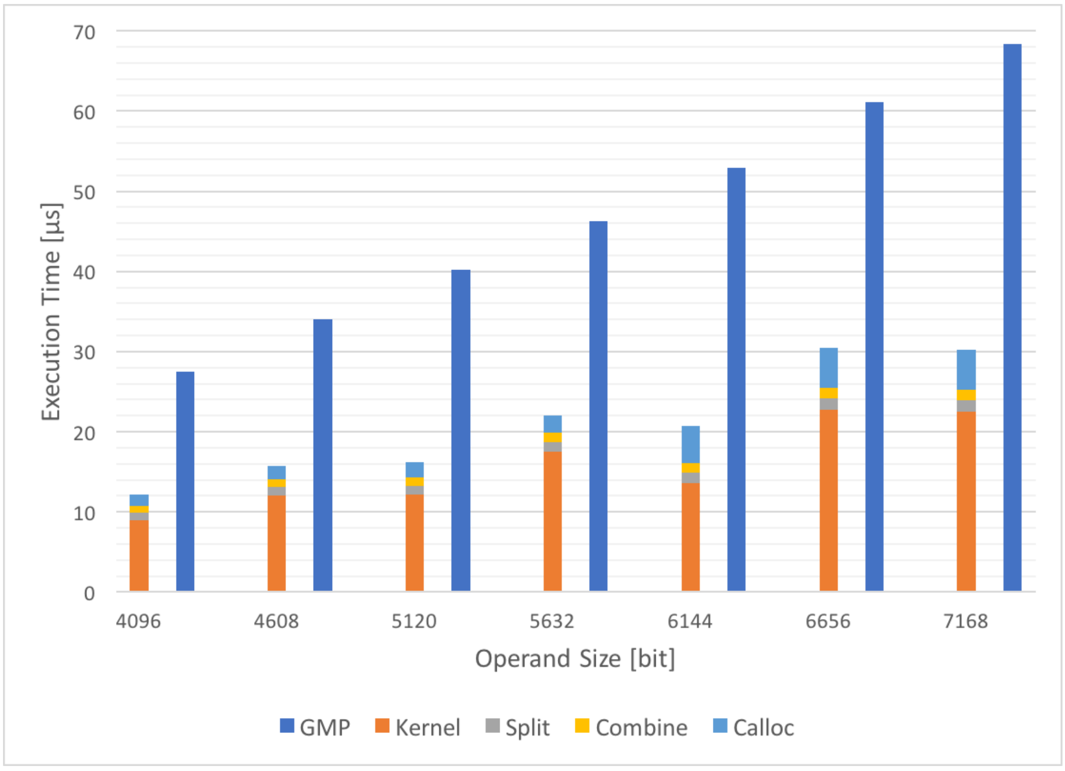


Figure 3.8: Execution time of multiplication function in proposed program and GMP 6.1.2 for 4,096- to 7,168-bit operands.

lates multiplication for 896-bit operands each time it is called (i.e., the module deals with multipliers and multiplicands of 896 bits at once). Therefore, for example, the processing times for 2,048- and 2,560-bit multiplication that satisfies 1,792 $(= 896 \times 2) < n \leq 2688$ $(= 896 \times 3)$ in n-bit multiplication are roughly equivalent.

The execution time of the proposed program increases significantly for 6,656-bit or higher operation. After this point, the total amount of data in arrays for the partial products and the converted 2^{28}-radix number exceeds 32 KB. Hence, it can be concluded that the delay could have been caused by the insufficient capacity of the L1 cache on the processor used in this experiment.

3.2.4 Conclusion

In this section, a multiplication program that can flexibly deal with variable-length operands with AVX-512F instructions and the reduced-radix representation was implemented and evaluated on the Knights Landing architecture. Furthermore, two methods for accelerating large integer multiplication were proposed. The first method is to implement a module specialized for fixed-length multiplication and repeatedly call this module to support flexible-length multiplication. This method allows the loop to be unrolled and optimized. The second method is to divide the partial product arrays to reduce stall as much as possible. Based on an evaluation, GMP is superior to the proposed program for 512-bit operands. However, for 1,024-bit or higher operands, a performance improvement in terms of execution time of up to approximately 2.5x compared with GMP was achieved. A speedup of the multiplication of large variable-length integers was also achieved.

When multiplying small operands, the cost of converting a number and allocating multiple arrays affects the total execution time. Moreover, although AVX-512 handles eight 64-bit integers with vector operations, it is not simply 8x faster than GMP, which uses scalar operations. Even taking into consideration the above-mentioned conversion and allocation cost, the total cost appears to be no more than eight times that of GMP. Therefore, the present study suggests that scalar operations are sufficiently fast and that vector instructions are not necessarily faster than scalar instructions. Nevertheless, experiments on the Xeon Phi Knights Landing processor showed that AVX-512F effectively accelerates large integer multiplication.

3.3 Multiplication Using AVX-512IFMA

AVX-512F instructions enable the execution of basic operations such as 512-bit-wide SIMD addition, multiplication, and floating-point fused multiply-add (FMA). Processors with the Cannon Lake microarchitecture enable the execution of AVX-512IFMA instructions. This allows integer multiplication and addition to be processed in one instruction.

Herein, the performance of large integer multiplication using the reduced-radix representation and AVX-512IFMA instructions is evaluated on a processor with the Cannon Lake microarchitecture. The Karatsuba method is additionally implemented in this section.

3.3.1 Karatsuba Multiplication

In this section, the Karatsuba multiplication algorithm [39] is implemented. Algorithm 3 shows the Basecase method, which is called inside the Karatsuba method.

Algorithm 3 BasecaseMultiply [37]

Input: $A = \sum_{i=0}^{m-1} a_i\beta^i, B = \sum_{j=0}^{n-1} b_j\beta^j$

Output: $C = A \cdot B := \sum_{k=0}^{m+n-1} c_k\beta^k$

```
1: $C \leftarrow A \cdot b_0$
2: for $j \leftarrow 1$ to $n-1$ do
3:     $C \leftarrow C + (A \cdot b_j)\beta^j$
4: end for
5: return $C$.
```

Algorithm 4 KaratsubaMultiply

Input: $A = \sum_{i=0}^{n-1} a_i\beta^i, B = \sum_{j=0}^{n-1} b_j\beta^j$

Output: $C = AB := \sum_{k=0}^{2n-1} c_k\beta^k$

```
1: if $n \leq n_0$ then
2:     Borrow process
3:     return BasecaseMultiply$(A, B)$
4: end if
5: $k \leftarrow \lceil n/2 \rceil$
6: $(A_0, B_0) := (A, B)$ mod $\beta^k$
7: $(A_1, B_1) := \lfloor (A, B) / \beta^k \rfloor$
8: $(A_2, B_2) := (A_0 - A_1, B_0 - B_1)$
9: $C_0 \leftarrow$ KaratsubaMultiply$(A_0, B_0)$
10: $C_1 \leftarrow$ KaratsubaMultiply$(A_1, B_1)$
11: $C_2 \leftarrow$ KaratsubaMultiply$(A_2, B_2)$
12: return $C := C_0 + (C_0 + C_1 - C_2)\beta^k + C_1\beta^{2k}$.
```

Algorithm 4 shows the Karatsuba multiplication kernel. The asymptotic computational complexity of the Karatsuba method is $O(n^{\log_2 3}) \approx O(n^{1.585})$ and that of the Basecase method is $O(n^2)$, where n is the number of operand words. Before KaratsubaMultiply is called for the first time, n is adjusted such that it is always even until n satisfies $n \leq n_0$. If the number of words in the multiplier and multiplicand is less than or equal to n_0 in Algorithm 3, BasecaseMultiply is called and the function returns. The essence of Algorithm 4 is as follows:

1. A and B are split into halves, denoted as A_0, A_1 and B_0, B_1, respectively,
2. the two lower parts, the two upper parts, and $(A_0 - A_1)$ and $(B_0 - B_1)$ are multiplied separately, and
3. the partial products are added at the appropriate position.

The Karatsuba method is based on the following equations.

$$(A_0 - A_1)(B_0 - B_1) = A_0B_0 + A_1B_1 - A_0B_1 - A_1B_0 \tag{3.1}$$

$$A_0B_1 + A_1B_0 = A_0B_0 + A_1B_1 - (A_0 - A_1)(B_0 - B_1) \tag{3.2}$$

Equation (3.1) shows the usual multiplication of AB when A and B are split into (A_0, A_1) and (B_0, B_1). In this expression, it is necessary to obtain four independent partial products, namely A_0B_0, A_1B_1, A_0B_1, and A_1B_0, to calculate AB. However, in equation (3.2), $A_0B_1 + A_1B_0$ can be obtained by calculating $(A_0 - A_1)(B_0 - B_1)$ after calculating $A_0B_0 + A_1B_1$. In other words, if the final additions are executed appropriately, it is possible to obtain AB through three multiplications.

64×82 in decimal format is used as an example. Figure 3.9 shows the 64×82 process for the Basecase method (left) and the Karatsuba method (right). With the Basecase method, 5,248 is obtained by performing four multiplications, namely 4×2, 6×2, 4×8, and 6×8. With the Karatsuba method, 5,248 is obtained by performing three multiplications, namely 4×2, 6×8, and $(4 \times 2 + 6 \times 8) - (6 - 4)(8 - 2)$. The Karatsuba method yields the correct results even if either or both of $A_0 - A_1$ and $B_0 - B_1$ are negative. Therefore, the Karatsuba method can be used to obtain the same result as that obtained using the Basecase method but with fewer multiplications. The Karatsuba method appears somewhat more complex for such a simple example. However, in computational terms, multiply instructions are

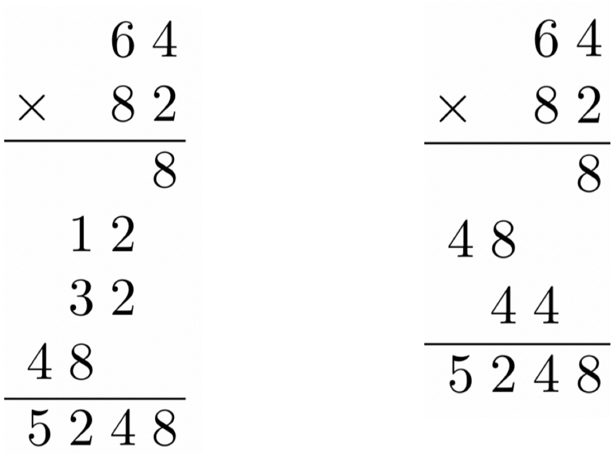


Figure 3.9: Multiplication using Basecase method (left) and Karatsuba method (right).

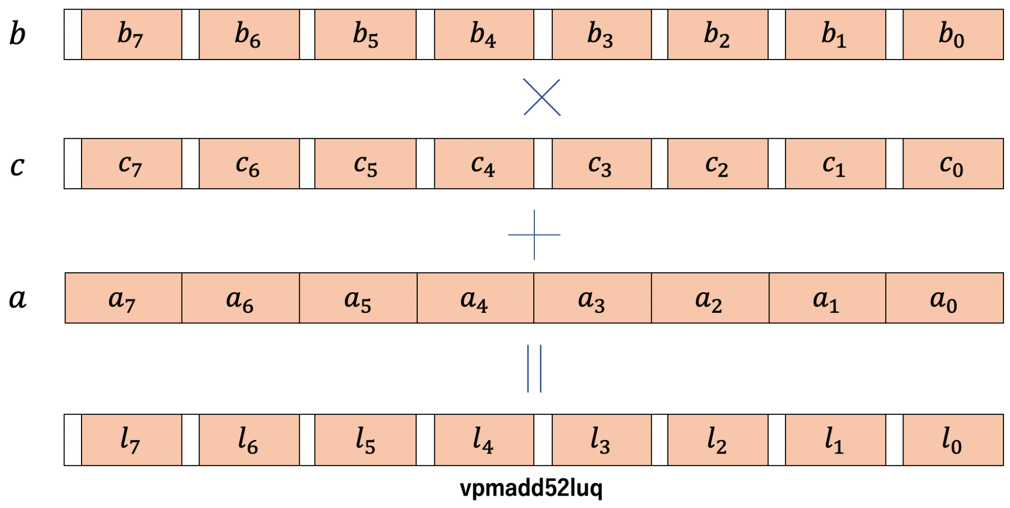
more expensive than addition and subtraction. Furthermore, for multi-precision integer multiplication, the number of multiply instructions is high. Thus, reducing this expensive operation is important for decreasing execution time.

Since lines 3, 8, and 12 in Algorithm 4 can be vectorized, AVX-512 instructions are used for these processes. A and B are in the 2^{52}-radix redundant representation and the arguments of BasecaseMultiply must be within the 52-bit representation. Therefore, before BasecaseMultiply is called, the borrows generated in $A_0 - A_1$ and $B_0 - B_1$ must be processed. Borrows need not be handled for non-subtracted arguments because no borrows occur for these arguments. Therefore, line 2 of Algorithm 4 can be skipped to avoid unnecessary processing in these cases. In the proposed implementation, because the length of operands A and B is pre-adjusted such that the highest word of A_1 and B_1 is zero, the case of $(A_0 - A_1)$ and $(B_0 - B_1)$ becoming negative is avoided as much as possible. If one of the two subtractions is expected to result in a negative value, the order of the subtraction can be switched to avoid the negative value and then $(C_0 + C_1 - C_2)$ can be changed to $(C_0 + C_1 + C_2)$ as appropriate.

3.3.2 Implementation

AVX-512 Instructions for Implementation

Before AVX-512IFMA can be executed, `vpmuludq` should be used to perform large integer multiplication. However, this instruction takes a 32-bit multiplier and a 32-bit mul-



b
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
c
c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0
a
a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
l_7 l_6 l_5 l_4 l_3 l_2 l_1 l_0
vpmadd52luq

Figure 3.10: Overview of `vpmadd52luq` processing.

tiplicand to make 64-bit partial products in parallel. AVX-512DQ instructions introduced `vpmullq`, which takes two 64-bit integer operands. However, `vpmullq` returns only the lower 64-bit part of a 128-bit integer obtained by multiplication. There is no way to obtain the upper 64-bit part. Therefore, AVX-512IFMA instructions offer the following three advantages over `vpmuludq` and `vpmullq`.

- They take operands larger than 32 bits.
- They calculate addition and multiplication in one instruction.
- They return both the low and high partial products.

AVX-512IFMA instructions are thus useful for multi-precision integer multiplication.

Figure 3.10 shows integer multiply-add operations using `vpmadd52luq`, an AVX-512IFMA instruction. The upper two registers b and c in this figure are the operands used for multiplication and the lower register a is the operand used for addition. Each element of b and c uses only the lower 52 bits as its operand; the remaining 12 bits are not used. Therefore, the white part in the figure is not involved in multiplication. The a used for addition is computed with 64 bits for each element.

The two intrinsic functions of the AVX-512IFMA instructions used in this research are the ones given in [27].

`_mm512_madd52lo_epu64(__m512i a, __m512i b, __m512i c)`

Calculate multiply-add in 512-bit-wide SIMD registers. This function is converted into `vpmadd52luq`. Multiply packed unsigned 52-bit integers `b` and `c` and get 104-bit intermediate products. Then, add 52-bit integer `a` and the 104-bit intermediate products and generate results. This function (instruction) returns the lower 52 bits of the result.

`_mm512_madd52hi_epu64(__m512i a, __m512i b, __m512i c)`

The same as `_mm512_madd52lo_epu64` except that it returns the upper 52 bits of the results. This function is converted into the `vpmadd52huq` instruction.

AVX-512 instructions are also used for the conversion to the reduced-radix representation and the multiplication kernel. The AVX-512BW instruction performs 8- and 16-bit integer operations. The AVX-512VBMI instruction performs 8-bit integer operations that cannot be performed by the AVX-512BW instruction. It consists mainly of 8-bit permutation instructions. These instructions are used specifically for element-by-element shift operations and permutation operations. Until the AVX-512BW and VBMI instructions were available, only 32- or 64-bit shift and permutation instructions could be executed using AVX-512F instructions. However, to convert from a 64-bit representation to a 52-bit representation and vice versa, operations with a small granularity of Byte or Word units are required. It was difficult to implement parallel conversion processing using AVX-512F instructions. To avoid the requirement of Byte- or Word-level granularity using AVX-512F instructions, gather and scatter instructions can be used. However, these instructions require non-continuous memory access, resulting in high latency. With the AVX-512BW and VBMI instructions, it is now possible to perform the conversion to the reduced-radix representation and the inverse conversion in parallel and efficiently.

The following four instructions are used in this study: `vpsrlvw` and `vpsllvw` for the right- and left-shift instructions, respectively, and `vpermw` and `vpermb` for the permutation instructions for Word and Byte units, respectively (16- and 8-bit operations, respectively).

Figure 3.11 shows an overview of the behavior of the `vpermw` instruction, a permutation instruction in AVX-512BW. The upper register is a 512-bit register that serves as the source of permutation data, and the middle register is an index register that indicates which data are taken. The permutation procedure performed by the `vpermw` instruction is

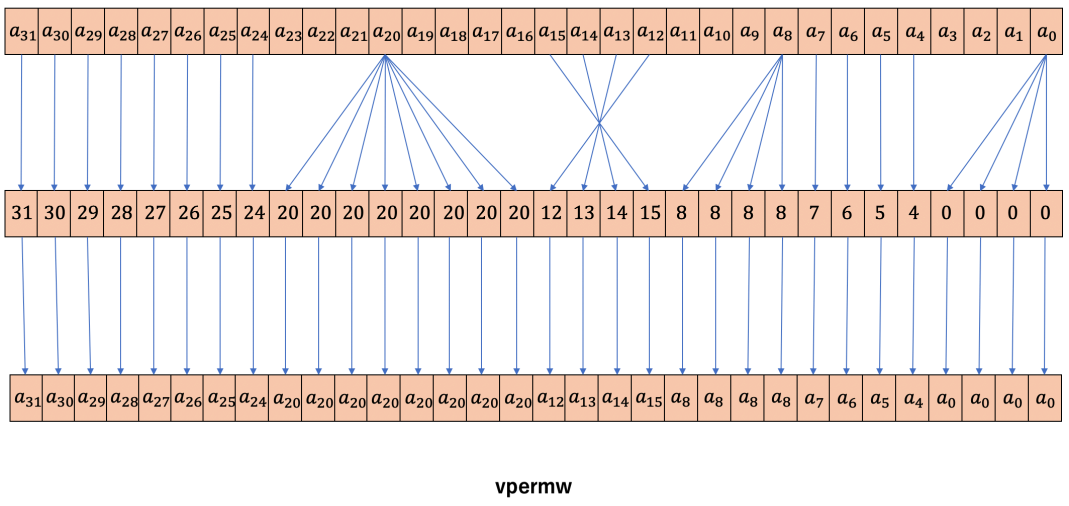

a31 a30 a29 a28 a27 a26 a25 a24 a23 a22 a21 a20 a19 a18 a17 a16 a15 a14 a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a0
31 30 29 28 27 26 25 24 20 20 20 20 20 20 20 20 12 13 14 15 8 8 8 8 7 6 5 4 0 0 0 0
a31 a30 a29 a28 a27 a26 a25 a24 a20 a20 a20 a20 a20 a20 a20 a20 a12 a13 a14 a15 a8 a8 a8 a8 a7 a6 a5 a4 a0 a0 a0 a0
vpermw

Figure 3.11: Overview of `vpermw` processing.

as follows:

1. divide the 512-bit data source into 32 16-bit parts $(a_0, a_1, \ldots, a_{31})$;
2. store a number from 0 to 31 that indicates a necessary element in each 16-bit space of the index register;
3. copy data a_j from the source register to each space of the destination register according to the index j specified in the previous step.

The AVX-512VBMI instruction `vpermb` operates with a granularity of eight bits with 64 divisions in the above procedure.

Figure 3.12 shows an overview of the behavior of the `vpsrlvw` instruction, a right-shift instruction in AVX-512BW. The upper register is a 512-bit source register and the middle register contains count values that indicate how many shifts are to be performed. Based on the number (from 0 to 15) stored in each element of the count register, the corresponding element of the data source is shifted to the right. The space vacated by the shift is filled with zeros. That is, the white portion of the destination register represents zero.

To use the four instructions above, the following intrinsic functions are used.

```
_mm512_permutexvar_epi16 ( __m512i idx, __m512i a)
```

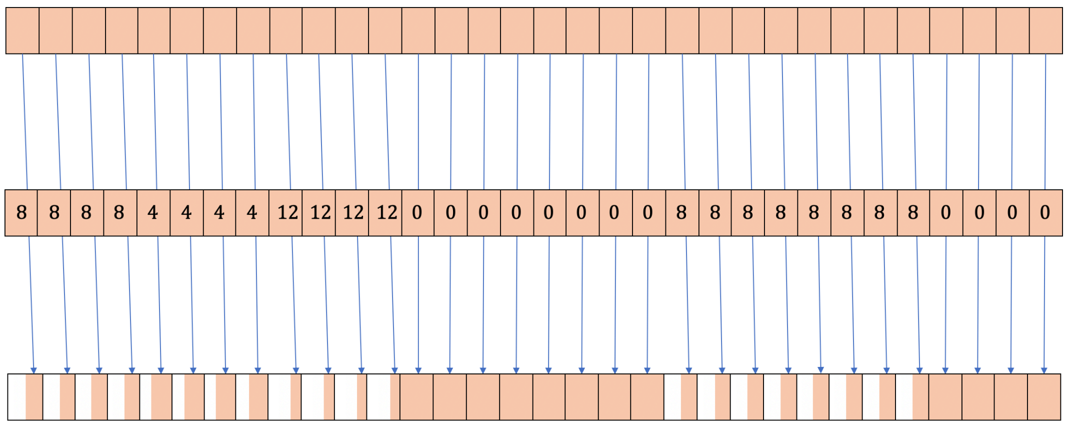


Figure 3.12: Overview of `vpsrlvw` processing.

Perform permutation operations on 16-bit integers using 512-bit-wide SIMD registers. This function is converted into a `vpermw` instruction.

`_mm512_permutexvar_epi8 ( __m512i idx, __m512i a)`

Similar to the above function, it performs permutation but with 8-bit granularity.

`_mm512_srlv_epi16 ( __m512i a, __m512i count)`

Right-shift each corresponding element of vector `a` by the numerical value stored in each element of vector `count`. This function is converted into the `vpsrlvw` instruction.

`_mm512_sllv_epi16 ( __m512i a, __m512i count)`

Same as `_mm512_srlv_epi16` except that each element of vector `a` is left-shifted. This function is converted into the `vpsllvw` instruction.

Implementation of Multiplication Kernel

Multi-precision integer multiplication is implemented using the Karatsuba method, as stated in subsection 3.3.1. Lines 6 and 7 of Algorithm 4 require modular arithmetic and division, respectively. However, since a large integer is expressed with 64-bit unsigned integer arrays, (A_0, B_0) is passed the start address of each operand and (A_1, B_1) is given the midpoint

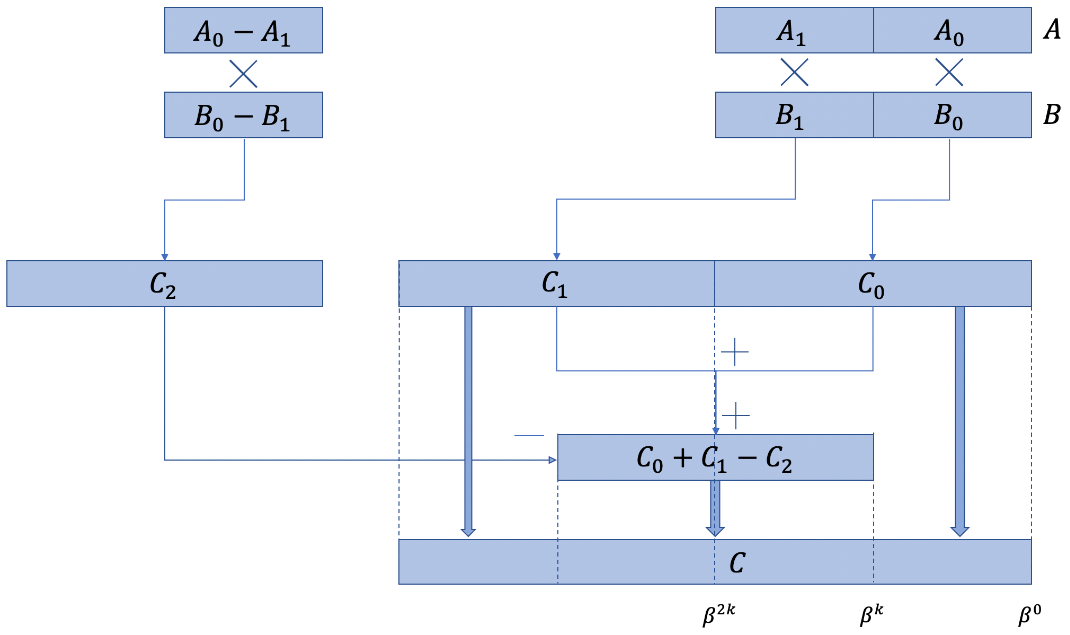


Figure 3.13: Calculation of AB large integer multiplication using Karatsuba method.

address of the arrays. Therefore, modular arithmetic and division can be performed by manipulating the address references.

Since subtractions are performed on line 8 of Algorithm 4 and vectorization is possible, these processes are vectorized. Borrows as well as carries can be accumulated because of the reduced-radix representation. Subtraction can thus be performed without concern for data dependencies. Unfortunately, AVX-512IFMA instructions ignore the upper 12 bits of the operand, which store the borrows. Hence, BasecaseMultiply cannot be used with the borrows without a cleanup done in advance. Line 2 is skipped in the case of non-subtracted arguments and BasecaseMultiply (i.e., $((A_0, B_0)$ and $(A_1, B_1))$) is performed. This avoids unnecessary processing. In this study, splitting was done only once. This is because as the number of divisions increases, the number of operands to be subtracted also increases, resulting in an increase in the ratio of cleanup processing to the total.

On line 12 of Algorithm 4, $(C_0 + C_1 - C_2)$ is calculated and each partial product is added to the appropriate place, as shown in Figure 3.13. Finally, the result C is obtained. In these processes, SIMD instructions are vectorized using AVX-512 instructions. As noted in subsection 3.3.1, the Karatsuba method requires fewer multiplications compared with the Basecase method.

3.3.3 Evaluation

Evaluation Environment

For evaluation, a host computer with an Intel Core i3-8121U (2.2 GHz, 2 Cores, 4 Threads, Cannon Lake microarchitecture) and 4 GB DDR4 memory running the Ubuntu Server 18.04.2 LTS operating system was used. All of the proposed experimental programs were implemented in the C language and compiled using `icc` version 19.0.4.243 with the `-O3 -Wall -xCANNONLAKE -mtune=cannonlake -std=c99` options. The comparison targets were programs in [33], the implementation in Section 3.2, and GMP version 6.1.2. These programs were also built with `icc`. In the evaluation, all programs were executed on a single core with a single thread. Multiplication was performed 3,000 times and the execution time was averaged. To generate the multiplier and multiplicand, the `rand` function was used. The system time was used as the random seed.

Comparison with Basecase Multiplication

As discussed, a large integer multiplication kernel was implemented here using the Karatsuba method. Although the amount of computation for the Karatsuba method is smaller than that for the Basecase method, the latter has shorter execution times if the operand size is small because the former uses Basecase multiplication several times. Therefore, the execution times of the algorithms are compared. The results of this comparison can be used to determine the optimal size for switching between the Karatsuba and Basecase algorithms. Indeed, GMP switches algorithms according to operand size.

Figure 3.14 compares the execution times for various operand sizes between the Karatsuba and Basecase methods with AVX-512F (interrupted due to the limitation of the 2^{28}-radix representation) and AVX-512IFMA. Since both methods use the reduced-radix representation, the execution time of the multiplication kernel is of concern here.

With the Basecase method, AVX-512IFMA is faster than AVX-512F at all sizes. With the Karatsuba method, the execution times for AVX-512IFMA and AVX-512F are similar for small sizes. Although AVX-512F is superior to AVX-512IFMA at 2,048 bits, AVX-512IFMA is faster at larger sizes. The Karatsuba method with AVX-512IFMA requires borrow processing according to line 2 in Algorithm 4. In contrast, this method with AVX-512F does not need borrow processing because `vpmuludq` takes all 32 bits of 32-bit operands whereas `vpmadd52l(h)uq` takes only 52 bits of 64-bit operands. The

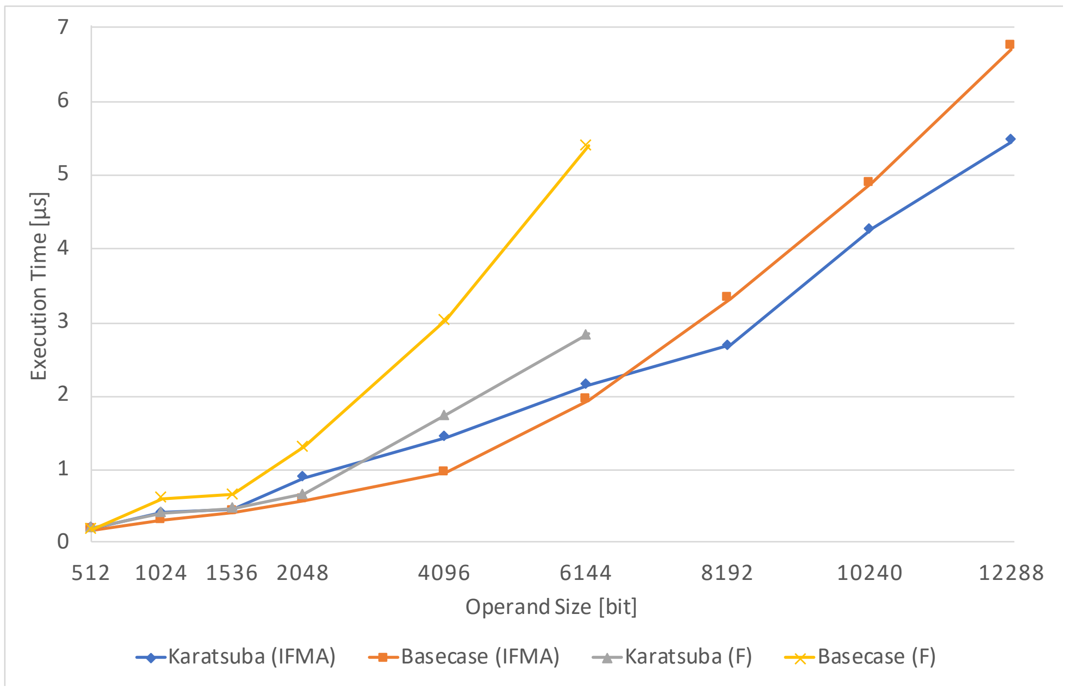


Figure 3.14: Execution time for Karatsuba and Basecase methods for 512- to 12,288-bit operands.

AVX-512IFMA-specific overhead for small operand sizes cannot be ignored, especially for operations faster than one microsecond. Furthermore, due to the overhead, the kernel with AVX-512IFMA divides one operand into two parts, whereas that with AVX-512F splits it into up to four parts. This is because the former increases the number of times that extra borrow processing is performed with increasing number of partitions. Therefore, there is also a difference between the two instruction sets in terms of actual complexity for a given operand. However, AVX-512IFMA deals with 416 ($= 52 \times 8$) bits in multiplication and addition in one instruction whereas AVX-512F deals with 224 ($= 28 \times 8$) bits in multiplication and requires `vpaddq` for addition. Thus, AVX-512IFMA is faster for larger sizes despite the overhead.

With AVX-512IFMA, the Basecase method was faster than the Karatsuba method for small operand sizes (from 2,048 to 6,144 bits). However, when the operand size was large (from 8,192 to 12,288 bits), the Karatsuba method was faster. Because the Basecase method has a computational complexity of $O(n^2)$ for multiplication and $O(n)$ for load/store, where n is the number of operand words, the execution time increases with operand size even if

AVX-512IFMA instructions are used. On the other hand, although the Karatsuba method reduces the complexity of multiplication to three-quarters that for the Basecase method, addition and subtraction with a computational complexity of $O(n)$ are executed on lines 8 and 12 of Algorithm 4. The load/store processes for addition and subtraction are also performed separately from the multiplication and thus the Karatsuba method requires more memory operations than does the Basecase method. In addition, since the Karatsuba method with AVX-512IFMA also performs extra borrow processing, the performance improvement rate decreases relative to the Basecase method for a given instruction.

In summary, with AVX-512IFMA, the Basecase method is faster when the size of the multiplier and multiplicand is small because the Karatsuba method cannot ignore the overhead due to the extra borrow processing, function calls, and load/store. For larger operands, however, the Karatsuba method is faster than the Basecase method. Based on the results of this experiment, the Basecase method should be switched to the Karatsuba method when the operand size is 7,168 bits or more.

Comparison with Related Works

The proposed implementation is compared with the program in [33] and the implementation in Section 3.2. Figure 3.15 shows a visualization of the differential execution times. The execution times for the implementation in Section 3.2 and the proposed method consist of Kernel, Split, and Combine. "Split" refers to the conversion from a 2^{64}-radix number to a 2^{52}-radix number in the proposed approach and from a 2^{32}-radix number to a 2^{28}-radix number for AVX-512F. "Combine" refers to the opposite conversion.

First, compared with AVX-512F, the proposed approach was faster at all operand sizes, with an improvement in performance of up to approximately 3.07x. This result indicates that AVX-512IFMA is superior to AVX-512F for large integer multiplication in terms of execution time. Furthermore, there are also differences in the execution times for Split and Combine between the proposed implementation and AVX-512F. The proposed program for conversion was implemented with AVX-512BW and AVX-512VBMI. Thus, AVX-512BW and AVX-512VBMI are also useful for accelerating the program.

Second, the program in [33], which was also implemented with AVX-512IFMA, is superior to the proposed program, although the execution times are similar. The AVX-512IFMA program in [33] is specialized for 1,024-, 2,048-, 3,072-, and 4,096-bit operands, and the code for Split, Combine, carry processing, and Kernel is in one module, which is

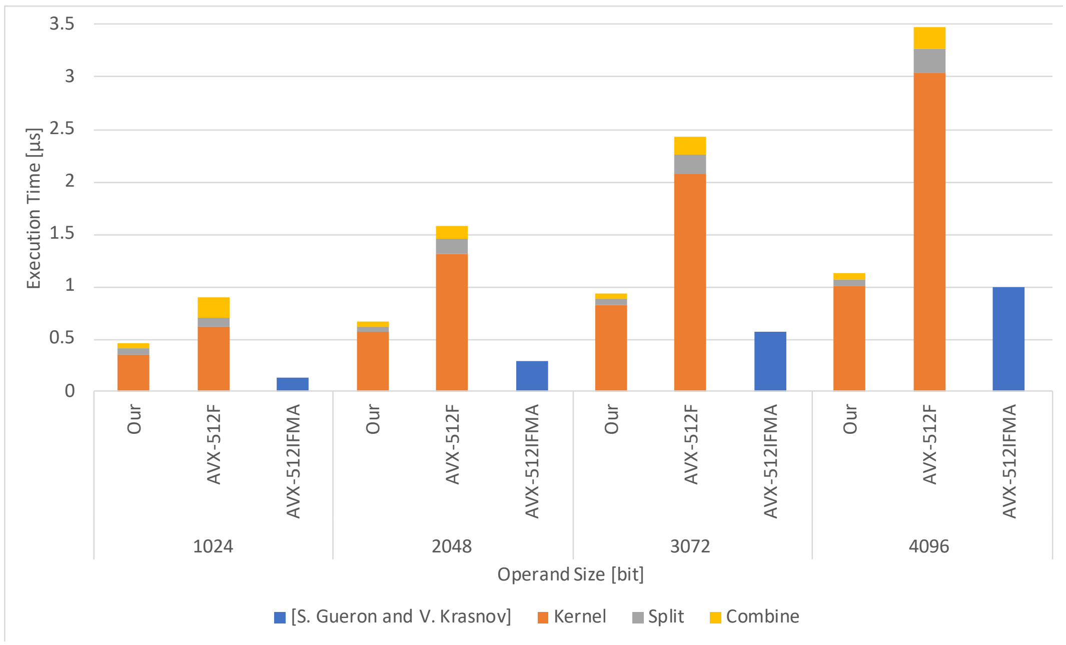


Figure 3.15: Execution time for Basecase method for 1,024- to 4,096-bit operands compared with related works.

similar to a monolithic kernel design. They are hard-coded for the target sizes. Therefore, the program in [33] was implemented using registers as much as possible to avoid memory access. In contrast, the proposed programs for Split, Combine, carry processing, and Kernel are implemented independently. These modules are called to support various operand sizes, which is similar to a microkernel design. Due to the modularization, there are costs associated with size-dependent branches, memory access overhead for calling modules, and load/store data. Especially for processing that takes less than one microsecond, the execution time of this overhead cannot be ignored. In light of this difference, a monolithic design for large integer multiplication with AVX-512IFMA is very fast for certain applications. In short, AVX-512IFMA has the potential to accelerate multiplication operations.

Comparison with GMP

The execution time of GMP and the proposed implementation for 1,024 to 12,288 bits was compared (Figures 3.16 and 3.17). Based on the results in Section 3.3.3, the proposed program is compared with GMP using the Basecase method in Figure 3.16 and the Karatsuba method in Figure 3.17. The execution times for 1,024 bits are almost equal and the

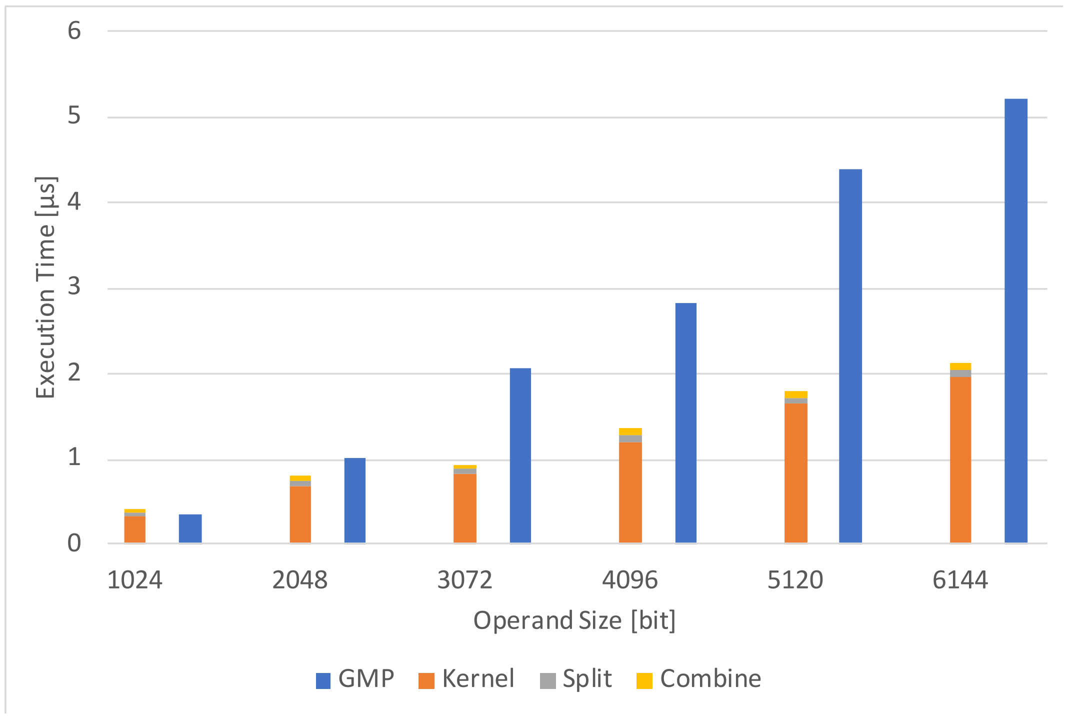


Figure 3.16: Execution time for multiplication function using AVX-512IFMA for 1,024- to 6,144-bit operands compared with GMP.

proposed program is superior to GMP for larger operand sizes. Although Kernel processing is faster than GMP at 1,024 bits, with the addition of Split and Combine processing, GMP is slightly faster. However, for other sizes, the proposed implementation is faster even with Split and Combine processing. An improvement of up to approximately 2.97x was obtained. GMP does not use SIMD instructions for multiplication, and thus this result suggests that vector instructions are faster than scalar instructions for large integer multiplication. This confirms that AVX-512IFMA instructions effectively speed up multi-precision integer multiplication.

3.3.4 Conclusion

In this section, multi-precision integer multiplication was speeded up by utilizing the reduced-radix representation and AVX-512IFMA. Furthermore, to speed up multiplication for large operand sizes, a multi-precision integer multiplication program was implemented using algorithms with the Karatsuba method and the conventional Basecase method. Since this

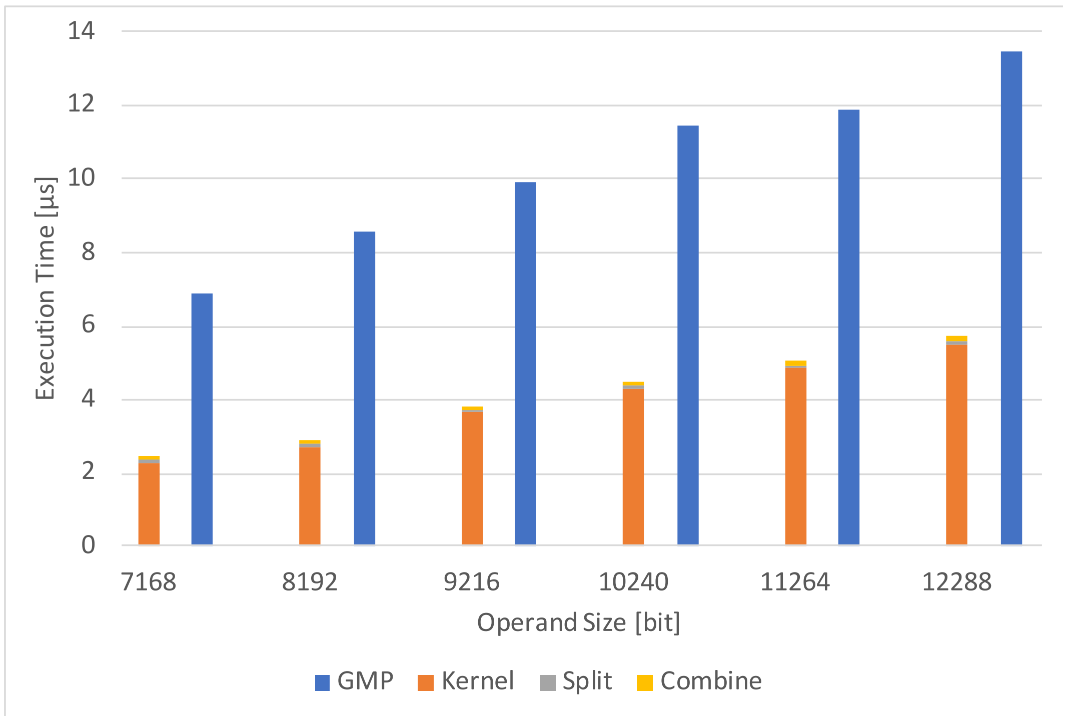


Figure 3.17: Execution time for multiplication function using AVX-512IFMA for 7,168- to 12,288-bit operands compared with GMP.

study focuses on general-purpose multiplication, the program was designed to handle various operand sizes with an emphasis on flexibility. The implemented program was evaluated on a processor with the Cannon Lake microarchitecture and its performance was compared with that of the implementation in Section 3.2, [33] and GMP in terms of execution time.

Compared with AVX-512F (Section 3.2), the proposed program was faster at all operand sizes; a performance improvement of up to approximately 3.07x was obtained. Furthermore, the results showed that AVX-512BW, AVX-512VBMI, and AVX-512IFMA effectively speed up calculation. The program in [33], which uses AVX-512IFMA, was faster than the proposed program at all sizes. This was mainly caused by the difference in the implementation of the multiplication kernel (monolithic design versus microkernel-like design). This experiment indicates that AVX-512IFMA instructions allow extremely fast multiplication for specialized applications. Finally, although the execution times for the proposed program and GMP were similar when the operand size was 1,024 bits, the proposed program was faster for multiplication with operand sizes of 2,048 bits or more. A performance improvement of up to approximately 2.97x was obtained. In summary, it was shown that AVX-512IFMA can speed up large integer multiplication.

Chapter 4

Large Integer Multiplication Using Arm SVE

The application of AVX-512 instructions to large integer multiplication was explored in the previous chapter, and it was found that these instructions are particularly effective for large operand sizes. Nowadays, Arm processors can also execute SIMD instructions with a vector length of 512 bits (Arm Scalable Vector Extension (SVE)) on A64FX processors [9]. However, it is not yet clear whether SVE, which has similar potential, is effective for such multiplication processes. Therefore, in the present chapter, SVE will be applied to large integer multiplication, and its performance will be evaluated to determine if it is as effective as AVX-512.

4.1 Related Works

Several studies have implemented and evaluated algorithms that use SVE. For example, SVE has been applied to sorting [40]. It was found that SVE sorting is faster than the GNU C++ Standard Template Library (STL) sorting method, with a performance improvement of about 5x for small-scale sorting and about 4x for large-scale sorting on an A64FX processor. SVE has also been applied to numerical computation. One study [41] applied SVE to the Basic Linear Algebra Subprograms (BLAS) [42] and compared its performance with that of Neon (Arm's 128-bit-wide SIMD instruction set). The results of simulator-based evaluations showed that SVE was faster in most cases for a 128-bit width. Furthermore, for SVE with a 2,048-bit width, a performance improvement of up to 17.77x over Neon was obtained, indicating that SVE accelerates these calculations. SVE has also been applied to deep neural networks (DNNs). One study [43] used SVE to vectorize the convolution, general matrix multiply (GEMM), and other processes required in image processing and benchmarked their system using tinyDNN neural network frameworks such as AlexNet [44] and ResNet [45]. Evaluation using an emulator indicated a speedup of more than 10x over the scalar version for all models.

SIMD instructions (especially AVX-512) have been applied to large integer operations and have been evaluated in real environments. For example, AVX-512F and IFMA have been applied to large integer multiplication in the previous chapter. For AVX-512F, the implemented program was run on the Intel Xeon Phi Knights Landing architecture and compared to GMP. A performance gain of up to approximately 2.5x was observed. For AVX-512IFMA, evaluations conducted on the Intel Cannon Lake microarchitecture showed that it was approximately 2.97x faster than GMP. One study [46] applied AVX-512 to modular multiplication. Compared with GMP, the implementation yielded a 3.2x higher throughput for a 1,024-bit calculation on an Intel Xeon Gold processor.

As shown above, SIMD instructions such as AVX-512 and SVE accelerate various computation tasks. In particular, AVX-512 instructions have been shown to be faster than scalar instructions for large integer multiplication. However, it is unclear whether SVE, which has similar potential, accelerates such multiplication. In this study, we apply SVE to large integer multiplication and evaluate the performance of our approach on the A64FX processor, which can execute 512-bit-wide SIMD instructions.

4.2 Multiplication Algorithm

4.2.1 Reduced-Radix Representation

First, we discuss the multiplier and multiplicand used in this study. Since this study is concerned with unsigned integers, we store the multiplier and multiplicand in 64-bit unsigned integer arrays.

In this study, the radix is set to 2^{56} (i.e., a 64-bit word is converted to a 56-bit word). This value was chosen for two reasons. The first is that it is convenient to perform the conversion using SVE instructions. Since $56 \times 8 = 448 = 64 \times 7$, converting seven 64-bit words yields eight 56-bit words. SVE can execute a 512-bit-wide (64 bits $\times$ 8 words) SIMD instruction. Seven words of data will not exceed the width of the vector after the conversion. During the conversion process, we can access memory in word units, which simplifies pointer handling. The second reason is that eight bits are sufficient for storing carries in this study. For a 56-bit representation, 255 ($= 2^8 - 1$) carries can be stored. Since this is equivalent to performing a multiplication of 256 ($= 2^8$) words, multiplication of up to 14,336 (= 56 bits $\times$ 2^8 words) bits can be computed without loss of carries. Note that the Karatsuba method [39] is likely to be faster for this number of bits, as indicated by experiments with AVX-512IFMA in the previous chapter. Therefore, it is reasonable to assume that there is little need to reduce the number of bits to fewer than 56 for Basecase multiplication. We thus adopt a 56-bit reduced-radix representation in this study.

4.2.2 Basecase Multiplication

In this chapter, Algorithm 3 is basically used for Basecase multiplication [37]. As noted in Section 4.2.1, $\beta = 2^{56}$ in this study. The main loop is represented as a single loop. However, a `for` loop that corresponds to the word length of A is actually executed inside that loop. Thus, the computational complexity of this method is $O(n^2)$.

Regarding SIMD instructions, the multiplication on the first line of Algorithm 3 and the multiplication and addition on the third line can be implemented as vector operations using SVE. The multiplication by β^j on the third line depends on the memory storage location and thus can be realized by appropriate pointer manipulation.

However, a problem arises when the reduced-radix representation is applied to ordinary multiplication instructions. AVX-512IFMA provides the instructions `vpmadd52luq` and `vpmadd52huq`, which take 52 bits for the multiplier and multiplicand and return the lower and upper 52 bits, respectively [19]. In other words, the target data are consistently in a reduced-radix representation from the input to the output. SVE, on the other hand, has no such instruction. Instead, it provides a SIMD instruction for ordinary 64-bit multiplication. There are two instructions in SVE, namely `MUL` and `MULH`, that return the lower and upper 64 bits of a 64-bit multiplication, respectively. We obtain a 112-bit partial product for a 56-bit calculation. For normal SVE multiplication instructions, the lower 64 bits and the upper 48 bits are returned, which makes it impossible to proceed with the calculation correctly since these values are no longer 2^{56}-radix numbers. This problem does not need to be considered in a full 64-bit calculation and is unique to the reduced-radix representation. Therefore, we need additional processing compared with AVX-512IFMA.

4.2.3 Shifted Basecase Multiplication

In this section, we discuss solutions to the problems related to the reduced-radix representation described in the previous section. The core of the problem is that when the normal 64-bit multiplication instruction is applied to a value represented by 56 bits, the lower word of the partial product will be larger than 56 bits (64 bits) and the upper word will be smaller than 56 bits (48 bits). Thus, we need to adjust these words to make them 56-bit values.

Algorithm 5 ShiftedBasecaseMul

```
Input: $A = \sum_{i=0}^{m-1} a_i\beta_r^i, B = \sum_{j=0}^{n-1} b_j\beta_r^j, \beta_r = 2^p, \beta = 2^{\max}, \beta_s = 2^{(\max-p)/2}, p < \max, \max - p$ is even
Output: $C = A \cdot B := \sum_{k=0}^{m+n-1} c_k\beta_r^k$
 1: $A' \leftarrow \sum_{i=0}^{m-1} a'_i\beta_r^i := \sum_{i=0}^{m-1} a_i\beta_s\beta_r^i$
 2: $B' \leftarrow \sum_{j=0}^{n-1} b'_j\beta_r^j := \sum_{j=0}^{n-1} b_j\beta_s\beta_r^j$
 3: $C \leftarrow 0$
 4: for $j \leftarrow 0$ to $n-1$ do
 5:     for $i \leftarrow 0$ to $m-1$ do
 6:         $c_L \leftarrow (a_i \cdot b_j) \bmod \beta_r$
 7:         $c_H \leftarrow (a'_i \cdot b'_j)/\beta$
 8:         $C \leftarrow C + c_L\beta_r^{i+j} + c_H\beta_r^{i+j+1}$
 9:     end for
10: end for
11: return $C$.
```

Algorithm 5 describes a method that solves this problem. β_r, β, and β_s are all assumed to be powers of two and `max` is the bit width of the multiplication instruction (64 in this study). The inputs A and B are the multiplier and multiplicand, respectively, both in reduced-radix representation ($p = 56$ in this study). To correctly obtain the upper part of the partial product in this algorithm, the original A and B multiplied by β_s should be calculated separately before the loop is entered. In the actual process, this is done by shift operations, so `max`$-p$ must be even. The calculation of the lower word of the partial product on line 6 is simple. The lower 56 bits are obtained from the remainder of the multiplication. For the upper word on line 7, we multiply the two pre-computed values described above. For example, since $p = 56$ and `max` $= 64$ in this study, a'_i and b'_j are 60 bits each and their lower four bits are zero. The partial product obtained by these multiplications is 120 bits, where the lower eight bits are filled with zeros (i.e., the product is shifted eight bits to the left). Therefore, by removing the lower 64 bits from the product, we get the correct upper 56 bits. Since c_L and c_H obtained by these operations are both 56-bit representations, we can proceed with the large integer multiplication.

The advantage of this algorithm is that it is relatively easy to implement with SIMD instructions. Although an instruction that returns the upper word of a partial product is required for multiplication, only two other basic instructions are required, namely a word-by-word shift and an AND operation. Therefore, it is likely that this technique can be applied to other architectures as long as instructions that correspond to `MUL` and `MULH` of SVE are available.

4.3 Implementation

4.3.1 Arm SVE

Arm SVE is an instruction set for the Arm AArch64 architecture [28]. In this instruction set, 32 scalable vector registers (Z registers) and 16 predicate registers (P registers) are available. These registers are similar to the zmm and mask registers in AVX-512, respectively. The critical difference is that the vector width for the AVX-512 instruction set is fixed at 512 bits, while the SVE scalable registers can be from 128 to 2,048 bits wide. This flexibility allows us to write bit-width-independent code and execute it on another SVE-supported processor without recompilation. The A64FX processor, the target processor in this study, is capable of executing 512-bit-wide SIMD instructions.

One implementation method for explicitly using SVE instructions is to use intrinsic functions. This allows appropriate processing by treating them in the same manner as C functions to use the desired SIMD instruction. When using these functions, we use special data types for SVE. For example, `svuint64_t` is used for a vector of unsigned 64-bit integers and `svfloat64_t` is used for a vector of 64-bit floating-point numbers. The predicate register should be of type `svbool_t`. For AVX-512 type declarations for zmm registers, the only information required is the bit width and whether the value is an integer or a floating-point number (e.g., `__m512i` for a 512-bit-wide integer vector). Therefore, we need to make specific type declarations according to the internal data type in SVE.

Next, we describe the main instructions used in this study. The intrinsic functions used here are listed below [21].

1. `svmul_n_u64_x(svbool_t pg, svuint64_t op1, uint64_t op2)`
2. `svmulh_n_u64_x(svbool_t pg, svuint64_t op1, uint64_t op2)`
3. `svadd_u64_x(svbool_t pg, svuint64_t op1, svuint64_t op2)`
4. `svlsl_n_u64_x(svbool_t pg, svuint64_t op1, uint64_t op2)`
5. `svand_n_u64_x(svbool_t pg, svuint64_t op1, uint64_t op2)`
6. `svext_u64(svuint64_t op1, svuint64_t op2, uint64_t imm3)`
7. `svtbl_u8(svuint8_t data, svuint8_t indices)`
8. `svreinterpret_u8_u64(svuint64_t op)`

The ”`_x`” at the end of some functions indicates that the return value of a Z register at the point where the predicate register is inactive (i.e., 0) is unknown. There are two other variations of the predicate register, namely ”`_m`” and ”`_z`”, which specify the behavior of the element in the register when inactive; they indicate that the element is to be merged with the first input and set to zero, respectively. The ”`_n_`” in the middle of some function names indicates that the trailing argument is a scalar value. In SVE's built-in functions, scalar values can be passed in place of vector values. When this happens, the scalar value is applied to all elements of the other vector operand. Finally, the ”`u64`” in a function indicates the data type of the element in the vector argument, which explicitly determines the type to be processed.

(1) and (2) perform 64-bit multiplication (`MUL` and `MULH`, respectively) and (3) performs 64-bit addition (`ADD`). In particular, `MULH` returns the upper 64 bits of a 128-bit partial product obtained by 64-bit multiplication. When A64FX became available, there were no such instructions in AVX-512; only AVX-512DQ (Doubleword and Quadword) included the instruction that returns the lower 64 bits (`vpmullq`). `MULH` allows us to perform larger integer multiplication with 64-bit-wide multiplications. As described in Section 4.2.1 regarding the `ADD` instruction, if the addition results in a carry, the carry is lost; the same is true for `vpaddq` in AVX-512. (4) and (5) show the left shift and AND operations

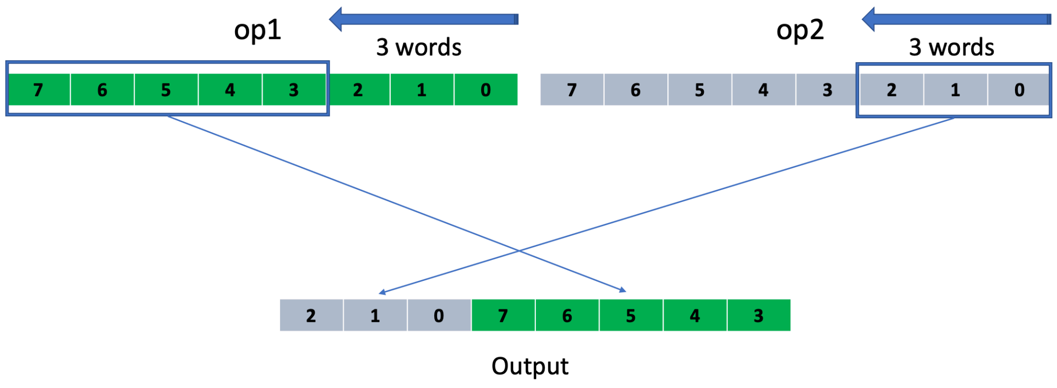


Figure 4.1: Overview of `EXT` when `op1`, `op2`, and an immediate value of 3 are passed.

(`vpsllq` and `vpandq` in AVX-512), respectively. Here, a scalar value is specified as the second operand in both cases. These instructions can also perform element-by-element shift and AND by passing a vector instead. (6) is an instruction that concatenates the elements from two Z registers. The position used for the concatenation depends on the last argument `imm3`. Figure 4.1 shows the behavior when 3 is passed to `imm3` of this instruction. In this case, the third and subsequent elements of `op1` are first taken out and filled in starting with the zeroth element in the output. Next, the remaining three words of the output are taken from the zeroth element of `op2` and filled in. The equivalent behavior of this instruction in AVX-512 is `valignq`. Furthermore, as the third argument ”`imm`” suggests, this value must be an immediate value (i.e., non-variable value). (7) is a shuffle instruction, which takes the element at the position specified in `indices` from `data` and stores it in the output. Since `u8` is specified here, we can shuffle in bytes. When this is executed on an Intel processor, the `vpermb` instruction is applicable, which requires the AVX-512VBMI (Vector Byte Manipulation Instructions) instruction set. Finally, (8) is used to explicitly convert the type of an element in a vector. Here, the interpretation is changed from 8 words of uint64 to 64 words of uint8. For AVX, the main requirement for the arguments to be passed to the intrinsic function is whether the elements are integers or floating-point numbers; in the case of integers, the number of bits is not a concern. However, as mentioned above, SVE intrinsics have a well-defined data type, so we need to use functions such as (8) to adapt them to the instructions.

As described above, within the scope of this study, the instructions available in AVX-

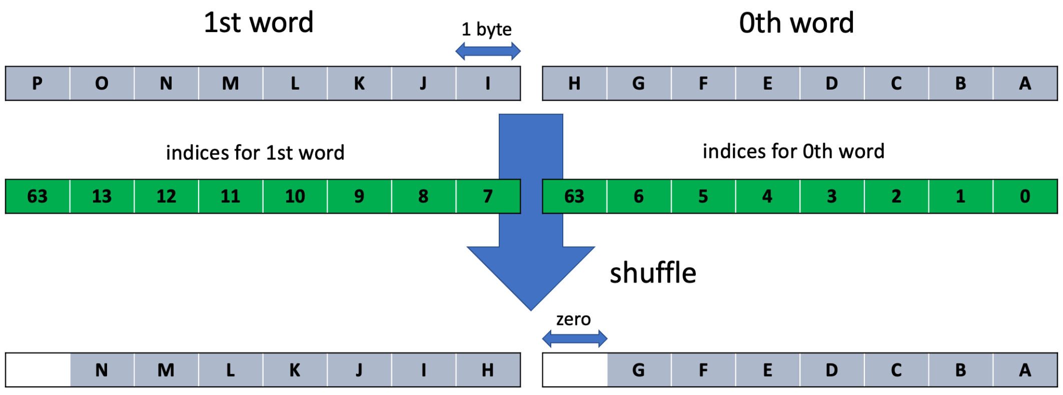


Figure 4.2: Overview of conversion from 64-bit representation to 56-bit representation. Two of the total eight words are shown. The values of the indices from 56 to 63 are all zeros, so for example, if 63 is specified, the data after shuffling will be zero.

512 are also available in SVE. In addition, SVE has the potential to speed up large integer multiplications because it can execute instructions such as MULH that do not exist in AVX-512.

4.3.2 Conversion of Reduced-Radix Representation

This section describes the implementation of conversion and inverse conversion to the reduced-radix representation described in Section 4.2.1. In this study, 64-bit words are converted to 56-bit words, which is a decrease of eight bits and can thus be achieved by byte-by-byte manipulations.

Figure 4.2 shows an overview of the conversion from a 64-bit representation to a 56-bit representation using SIMD instructions. Although eight words are processed simultaneously (the vector length is 512 bits), this figure shows only two words for simplicity. The gray data are the original and converted data. The white area is filled with zeros. One block represents one byte. The green data are the indices required by the `TBL` instruction ((7) described in Section 4.3.1). First, only seven words should be loaded from the original data, since the data after conversion to the reduced-radix representation are exactly eight words (the width of the vector), as described in Section 4.2.1. To achieve this, we utilize the predicate register. By setting the first seven words of this register to active (i.e., 1), we read only the active words and set the rest to zero. To create this predicate register data,

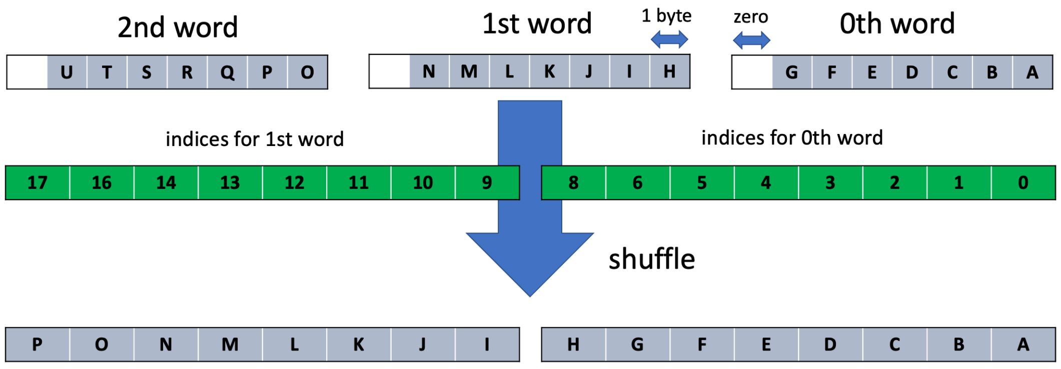


Figure 4.3: Overview of conversion from 56-bit representation to 64-bit representation. Two of the total eight words are shown.

several `enum svpattern` variables are provided for SVE programming. For our purposes, we created the predicate register using `SV_VL7` through the `svptrue_pat_b64` function. The trailing word of the data obtained by this process is zero. However, the data are used to fill the top byte of each word with zeros after conversion to a reduced-radix representation. Next, to allow shuffling on a byte-by-byte basis, we use `svreinterpret` to allow the data in the vector to be treated as uint8. We then use the `TBL` instruction to move the data to a 56-bit representation of each word and finally store the data. By following these steps, we can perform vector parallel conversion with SVE. Since AVX-512 also has mask instructions, this conversion technique is feasible.

We next describe the inverse conversion from a 56-bit representation to a 64-bit representation. Figure 4.3 shows an overview of the process. As shown in the figure, the policy of moving data to the appropriate position in bytes using the `TBL` instruction remains the same. The obvious difference is the way the last word on the output side is handled. The final word should normally be zero because the data become seven words when reverted to the 64-bit normal representation. However, as in the case of loading during a 56-bit conversion, the predicate register can be used to store the data as zeros in the corresponding part. Thus, we can proceed with the conversion in parallel by SVE.

4.3.3 Multiplication Kernel

Here, we describe our implementation of the multiplication kernel. Since SVE does not have special instructions such as AVX-512IFMA, and instructions are provided to process ordinary 64-bit multiplication in parallel, we implemented the Algorithm 5 method. In this study, we target arbitrary-precision integer multiplication so the design is as flexible as possible (unlike an implementation for fixed-length multipliers and multiplicands). Therefore, we implemented a fixed-length multiplication module that matches the vector length. This module can be called repeatedly to accommodate long operand lengths. Since the SVE of the A64FX processor is 512 bits wide, we implemented a module that performs an eight-word multiplication.

First, we describe the SVE implementation of the Algorithm 5-specific processing. The first and second lines of processing are multiplications by powers of two, so they can be realized by left shift operations. In this study, we left-shift each word by four bits with the `LSL` instruction ((4) in Section 4.3.1) to the Z register containing the multiplicand A to obtain A'. Since some SVE intrinsic functions allow passing scalar values as operands and `MUL` and `MULH` also allow this, for B we pass b_j shifted by four bits to the left. Next, we discuss the computation of the partial product c. On line 6 of Algorithm 5, c_L takes the lower 56 bits of the 112-bit partial product. We compute this with the `MUL` instruction with the usual operands a_i and b_j. Since the obtained product is 64 bits, the `AND` instruction of SVE is used to AND each word with $2^{56} - 1$ and extract 56 bits. For c_H on line 7, we need the upper 56 bits of the partial product. We perform a `MULH` calculation with operands a'_i and b'_j shifted to the left in advance. In Algorithm 5, although the modular operation is performed by β, since the `MULH` instruction automatically truncates the lower 64 bits of the partial product, we can obtain the upper 56 bits of the product without additional processing. With the above processing, latency is kept low because the shift and AND instructions in this algorithm can be processed without using memory.

A straightforward implementation of Algorithm 5 would involve memory access, especially for the processing on line 8. Since the computational complexity of the Basecase multiplication is $O(n^2)$, it is expected that accessing memory for that amount of time would result in enormous latency. However, in this implementation, the sizes of the multiplier and multiplicand passed to the module are known in advance, so we can customize our implementation. Figure 4.4 shows an overview of the Basecase multiplication kernel im-

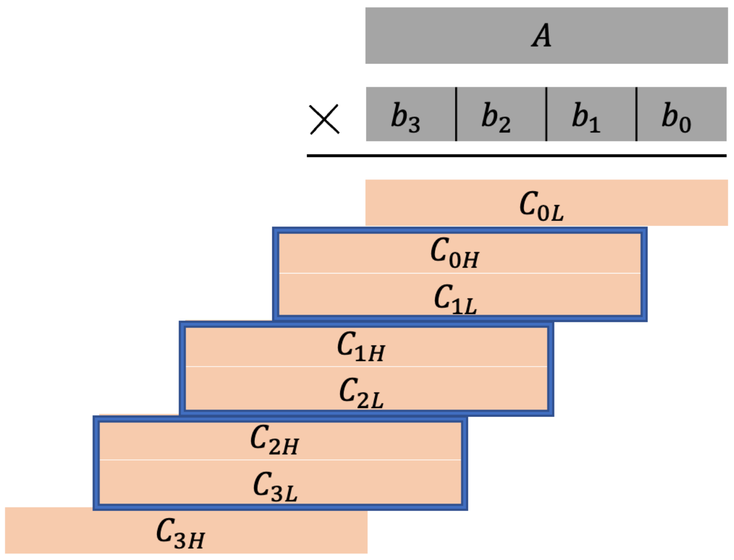


Figure 4.4: Overview of our implementation of Basecase multiplication kernel. In this figure, both the multiplier and multiplicand are four words.

plemented in this study. The two gray blocks are the input multiplier and multiplicand and the orange blocks below show the partial products. Although the vector length is shown here as four words for simplicity, the actual implementation is calculated with eight words to match the SVE of the A64FX processor. C_i is a partial product of $A \times b_i$. The subscripts L and H indicate a result of multiplication obtained by `MUL` and `MULH`, respectively.

Note that the partial products can be grouped except for those of the first and last multiplications. The partial products in the three groups outlined in blue in the figure $((C_{0H},\ C_{1L}),\ (C_{1H},\ C_{2L}),$ and $(C_{2H},\ C_{3L}))$ are stored in the same location. They can be simply added using SIMD operations. Therefore, it is convenient to calculate these two partial products at the same time.

Also note that in the example in this figure, the length of the result of the module does not always exceed eight words. The first and last partial product vectors (C_{0L} and C_{3H}) can cover just that eight-word range. Therefore, we accumulate the partial products in these two vectors until we finish the computation in the module. Specifically, we first load the already computed partial products in memory, which are in the same range as C_{0L} and C_{3H}, into the two Z registers. Next, we calculate C_{0L} and C_{3H} and add them to the loaded values. Then, we calculate C_{0H} and C_{1L} and add them together to obtain a new partial product. After the partial product is adjusted to the appropriate position, it is added to C_{0L} and C_{3H} and accumulated. This procedure is also performed for $(C_{1H},\ C_{2L})$ and $(C_{2H},\ C_{3L})$. Finally, after accumulating all the products, we store C_{0L} and C_{3H} at their original locations in memory.

For the implementation of adjusting partial products to the proper position of C_{0L} and C_{3H} described above, we used a Z register with all elements set to zero and the `EXT` instruction. For example, for the partial product of $C_{0H} + C_{1L}$, since it is shifted one word to the left relative to C_{0L}, we execute `EXT` with the zero register in `op1`, the partial product in `op2`, and 3 in `imm3`. Note that here the vector length is assumed to be set to 4, so an immediate value of 3 is correct. The partial product is shifted to a position where it can be added to the upper three words of C_{0L}. On the other hand, for the accumulation for C_{3H}, with the partial product in `op1`, the zero register in `op2`, and 3 in `imm3`, we obtain the remaining accumulated word. Thus, although this implementation requires the execution of some additional instructions for a word-by-word shift, this process does not access memory and thus does not incur load/store costs.

The implementation of a fixed-length multiplication module has some advantages. We know the size of the inputs and outputs, so we can unambiguously unroll the loop process. Furthermore, the only memory access that occurs when processing the module is the loading of the operands and source data and the storing of the final partial product. This minimizes the latency caused by memory access. Unfortunately, because the amount of shift specified in the `EXT` instruction must be an immediate value, our implementation assumes that an eight-word multiplication is performed (i.e., only for 512-bit-wide SIMD instructions). Therefore, we cannot take advantage of the vector length flexibility of SVE. However, if it were possible to specify the shift amount in this instruction by a value in a register, we would be able to implement a more flexible implementation.

Once all module calls are completed and the final product is obtained, we process the carries accumulated in the reduced-radix representation. Because of the data dependencies between adjacent words and the possibility of carry propagation, we perform this processing sequentially with scalar instructions.

4.4 Evaluation

4.4.1 Evaluation Environment

In this study we ran our program on an A64FX [9] (Arm v8.2-A + SVE) processor. The operating system was Red Hat Enterprise Linux 8. We compiled our experimental programs implemented in the C language with `fcc` version 4.8.0 in Fujitsu Development Studio. We always specified `-Kfast` as a compile option. Two binaries were created by specifying either `-Nnoclang` (trad mode) or `-Nclang` (clang mode). The comparison target was GMP [15] version 6.2.1. This library was also built with `fcc`. However, since GMP could not be compiled with the `-Nnoclang` option for compatibility reasons, we built it in only clang mode. We performed multiplications 5,000 times using the proposed method and GMP and averaged their execution times. We set the multiplier and multiplicand to random numbers (the system time was used as the random seed).

4.4.2 Comparison of Execution Times

In this section we evaluate the run times of our program (trad and clang modes) and GMP for various operand sizes. We specified the multiplier and multiplicand sizes to be from 1,024 to 14,336 bits.

Figure 4.5 shows a comparison of the execution times for various operand sizes (1,024 to 7,168 bits). The vertical axis is the execution time (μs) and the horizontal axis is the number of bits of the multiplier and multiplicand. For the proposed method, clang mode is marginally faster at 1,024 bits and comparable at 2,048 bits and trad mode is faster for operands larger than 3,072 bits. GMP is clearly faster at 1,024 bits and slightly faster than the proposed method at 2,048 bits. However, above 3,072 bits, our approach outperforms GMP. The reason GMP is faster for smaller sizes is because of the overhead of using SVE instructions. Moreover, to use these instructions, a multiplier and multiplicand must be converted to the reduced-radix representation, which accounts for a large percentage of the conversion time.

A comparison of execution times for operand sizes of 8,192 to 14,336 bits (the upper limit) is shown in Figure 4.6. Our implementations are faster than GMP in all cases in this range, with trad mode being faster than clang mode. These results show that large integer multiplication performed using SVE is faster than that performed using GMP, especially

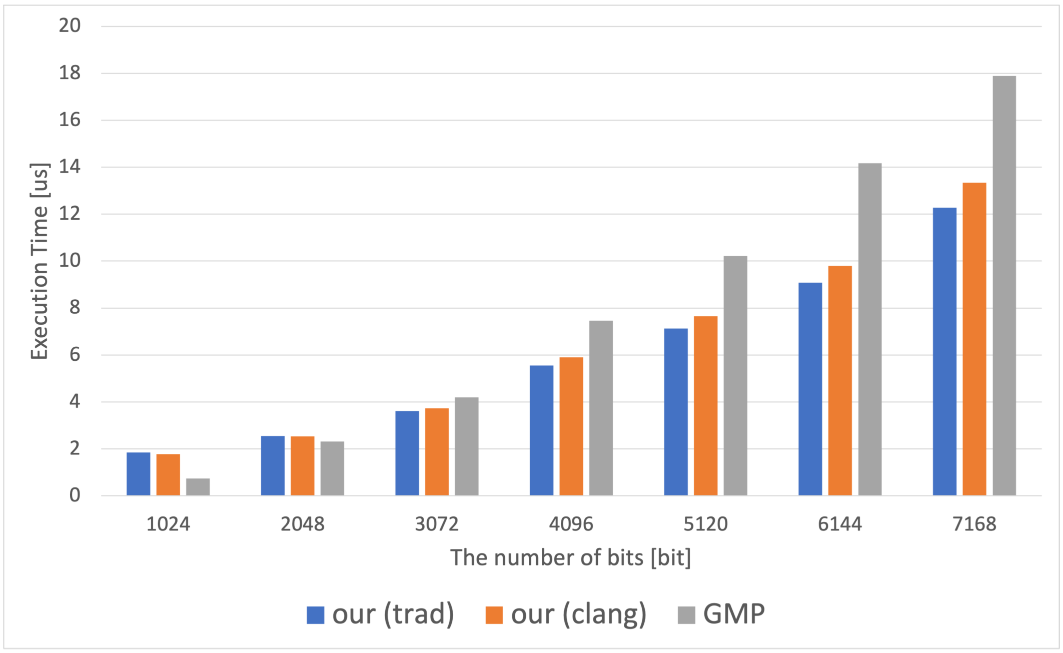


Figure 4.5: Execution times of proposed method and GMP for 1,024-bit to 7,168-bit operands.

when the operand size is large. Regarding the percentage of performance improvement over GMP, the trad mode and clang mode programs showed gains of up to 36% and 31%, respectively. GMP is implemented with scalar (not SIMD) instructions and is highly optimized for large integer multiplication. Therefore, these results indicate that SVE (SIMD) instructions accelerate such multiplication.

SVE on Arm processors does not provide as much performance improvement as does AVX-512IFMA on Intel processors. One reason for this is the difference in instruction latency. The latency of the AVX-512IFMA integer multiply-add instructions (`vpmadd52huq` and `vpmadd52luq`) is 4 cycles on an Ice Lake microarchitecture [27]. On the other hand, for SVE, the latency is 9 cycles for `MUL` (`MULH`) and 4 cycles for `ADD` on the A64FX architecture [47]. Thus, the latency is 3.25 times higher for SVE than for AVX-512IFMA in the calculation of the partial product at a single location. Another factor is the number of instructions. AVX-512IFMA performs multiplication and addition in a single instruction, whereas SVE computes them separately. In addition, we need to perform additional AND and shift operations to apply Algorithm 5. Therefore, when calculating with SVE, more instructions are required to be executed compared with those for AVX-512IFMA, which af-

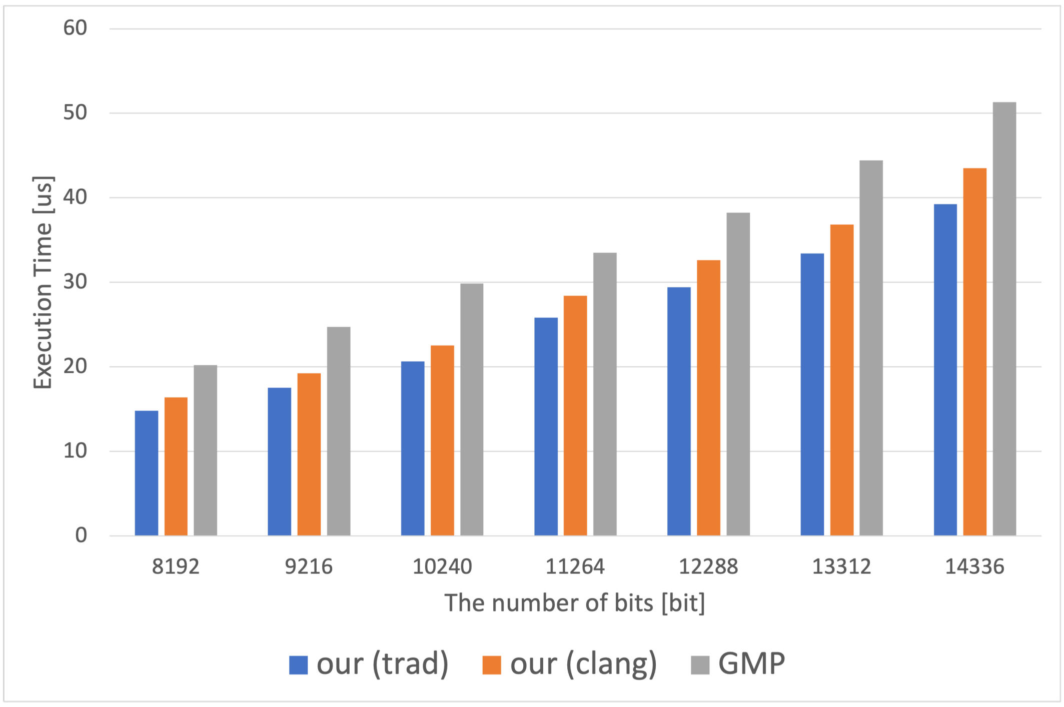


Figure 4.6: Execution times for proposed method and GMP for 8,192-bit to 14,336-bit operands.

fects the execution time. If instructions such as AVX-512IFMA were implemented in SVE, our approach would be faster since it would no longer be necessary to use the Algorithm 5 technique.

The difference between the execution time of our program and that of GMP decreases with increasing operand size. As the operand size increases, GMP switches to an asymptotically faster multiplication algorithm such as the Karatsuba method or the Toom–Cook method [48]. GMP thus algorithmically speeds up as the operand size increases. Our implementation, on the other hand, uses the Basecase algorithm consistently and thus the difference in execution time becomes smaller with increasing operand size. However, since our program is still superior in the range of operand size in this study, the SVE instruction set accelerates large integer multiplication.

GMP is faster when the multiplier and multiplicand are small because our implementation requires processing that is specific to SIMD instructions. When performing large integer multiplication, we convert the reduced-radix representation before and after computing the multiplication kernel and allocate memory to store the representation values.

Table 4.1: Comparison of total execution time and kernel execution time for multiplication for various operand sizes.

# of bits	trad (all)	trad (kernel)	ratio	clang (all)	clang (kernel)	ratio
1024	1.84	0.64	0.35	1.78	0.64	0.36
2048	2.54	1.37	0.54	2.54	1.42	0.56
3072	3.61	2.34	0.65	3.73	2.51	0.67
4096	5.55	4.27	0.77	5.90	4.64	0.79
5120	7.13	5.81	0.82	7.65	6.43	0.84
6144	9.08	7.78	0.86	9.80	8.51	0.87
7168	12.28	11.00	0.90	13.34	12.16	0.91
8192	14.82	13.47	0.91	16.37	14.98	0.92
9216	17.51	16.27	0.93	19.22	18.09	0.94
10240	20.62	19.24	0.93	22.54	21.52	0.95
11264	25.82	24.30	0.94	28.42	27.28	0.96
12288	29.39	27.54	0.94	32.63	31.37	0.96
13312	33.40	31.31	0.94	36.83	35.80	0.97
14336	39.23	37.47	0.96	43.51	42.87	0.99

However, these tasks are unnecessary for scalar processing (such as that used by GMP). This additional processing increases the execution time. Table 4.1 compares the execution time for the entire multiplication function and the kernel only. As shown, the kernel accounts for a small percentage of the total execution time when the number of bits is small; the percentage increases as the operand size increases. Therefore, in situations where the input and output are known in advance to be in a 56-bit representation, our approach can outperform GMP even for small operand sizes.

Referring to the results for 1,024 bits in Table 4.1, the execution time for the kernel alone is the same for trad and clang modes, but the overall processing time is faster for clang mode. Similarly, for the results for 2,048 bits, the kernel alone is faster for trad mode, even though the overall execution time is the same for the two modes. Thus, the program compiled in trad mode is overall the fastest in our performance evaluation, with the program compiled in clang mode being faster for some operations. Figure 4.7 compares the processing time for programs compiled in trad mode and with a combination of trad and clang modes (hybrid program). For the hybrid program, we specified trad mode for the multiplication kernel and clang mode for everything else based on the results in Table 4.1. This comparison shows that the hybrid program is faster than that compiled in trad mode,

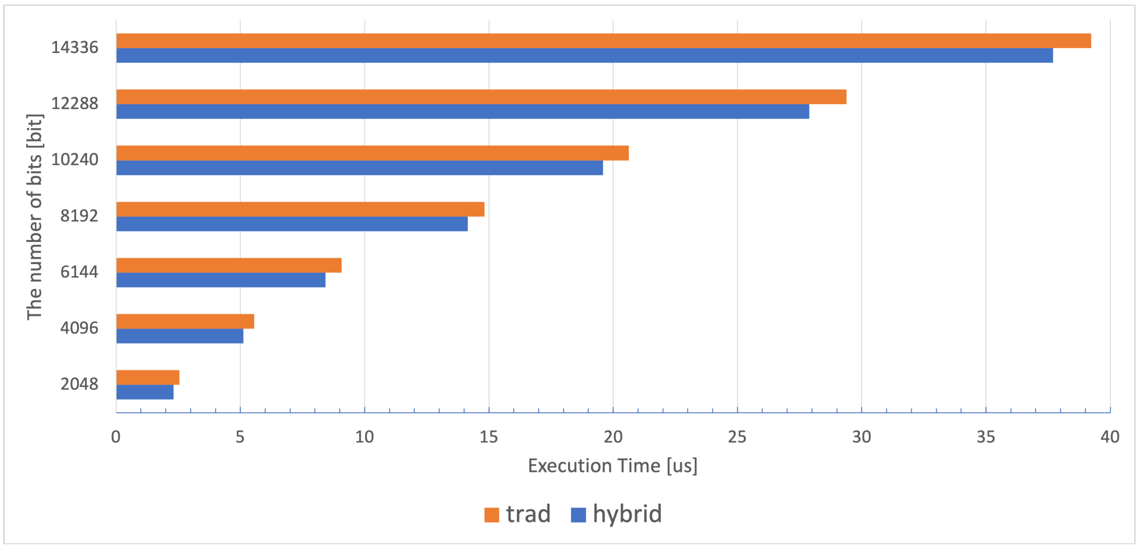


Figure 4.7: Comparison of integer multiplication execution time for programs compiled in trad mode and with combination of trad and clang modes (hybrid).

which was faster than GMP in Figures 4.5 and 4.6. This result indicates that the reason why the program compiled in trad mode was faster than that compiled in clang mode in most cases was because it was more compatible with the multiplication kernel, which was the dominant factor in the total execution time.

In summary, the SVE instructions enabled faster processing of large integer multiplication than that achieved by GMP. The execution time depended on the compilation mode, with trad mode found to lead to a shorter total processing time. The program compiled in clang mode was still faster than GMP.

4.5 Conclusion and Future Works

This study aimed to accelerate large integer multiplication for Arm processors. We used 512-bit-wide SVE instructions of the A64FX processor to speed up multiplication. We used the reduced-radix representation technique, which reduces the number of bits per word to accumulate carries, in order to increase vector parallelism. In addition, we proposed and implemented an algorithm that allows Basecase multiplication to be computed using a multiplication instruction that is longer than the bit length of a word in the reduced-radix representation. In the performance evaluation, we compared the running time of the implemented program with that of GMP, an arbitrary-precision arithmetic library. Even though GMP was faster for operands up to 2,048 bits, our SVE implementation was faster for larger operand sizes for both trad and clang compile modes. The performance gains were up to 36% in trad mode and up to 31% in clang mode. The above results show that SVE instructions have the potential to speed up large integer multiplication.

The most basic multiplication algorithm, Basecase multiplication, was implemented in this study. Faster multiplication algorithms such as the Karatsuba method [39] and the Toom–Cook [48] method will be implemented and evaluated for a large range of operand sizes in future studies.

In addition, the SVE2 instruction set has been introduced [49]. One of the instructions in this set targets large integer arithmetic and includes instructions that perform addition with a carry (`ADCLB` and `ADCLT`). These instructions may solve the carry problem that arises when SIMD instructions are used. Furthermore, once this problem is solved, we will be able to perform large integer multiplication without the reduced-radix representation, thus eliminating the need for conversion processing. We will implement a method combines SVE and SVE2 instructions and compare its performance with the method that uses only SVE and GMP.

The A64FX processor can execute 512-bit-wide SIMD instructions, whereas SVE instructions can handle up to 2,048 bits. A larger bit width should allow faster calculations, especially when the multiplier and multiplicand sizes are large. Therefore, we will evaluate the performance of our approach on processors that can execute instructions with larger bit widths.

Chapter 5

Large Integer Division Using Intel AVX-512

Since division, which is one of the most basic arithmetic operations, is much more expensive than multiplication in terms of computation time, speeding up division processing can be expected to yield important benefits. Currently, general x86 processors support up to 128-bit integer division instruction sets that concatenate two 64-bit registers and divide them by a 64-bit integer for scalar instructions. SIMD instructions are useful tools for fast large integer division computation because such instructions can process multiple data in parallel, and thus have the potential to perform various calculations faster than scalar instructions. Furthermore, since one way to reduce division costs is to replace the division operation with a multiplication operation, AVX-512IFMA greatly increases the possibility of speeding up both the division and multiplication of large integers.

With the above background in mind, this research aimed to accelerate large integer division using SIMD instructions, primarily AVX-512IFMA. To achieve this, several extant algorithmic calculations were implemented.

5.1 Related Works

The most basic division algorithm, introduced by Knuth [50], adjusts quotients one-by-one based on a candidate quotient and the remainder derived from it. One process that is asymptotically faster than classical division instructions is divide-and-conquer [51], which are algorithms that split divisors and dividends into several parts, and then recursively divide each part. These algorithms, which are particularly effective when both the divisor and dividend are somewhat long, also work well with large integer multiplications that are asymptotically fast because the quotient obtained by the recursive call is also a large integer, which means it can more easily incorporate the Karatsuba multiplication algorithm [39].

The first study in which the recursive method was incorporated achieved a speed level that was more than twice that of classical division when the word length was long. However, later, Burnikel's study pointed out that although this method is fast, it does not always yield the correct quotient [52]. Instead, Burnikel proposed a revised divide-and-conquer algorithm in which division was performed in a way that made it possible to obtain the correct quotient. They showed that a division of about 832 bits (250 decimal digits) could be computed on a SPARC architecture about twice as fast as could be achieved via the basic algorithm.

For calculating the divisions of very large integers, the Newton–Raphson method, which calculates an approximation of a reciprocal of a divisor by iteration and then multiplies it by a dividend to obtain a quotient, is effective. Additionally, by using the Karp and Markstein trick [53] in this method, the final quotient can be obtained without calculating the reciprocal in the last iteration. Another approach is the Barrett reduction process [54], which is similar to the Newton–Raphson method that mainly uses multiplication instead of division to obtain a quotient and remainder. More specifically, it calculates the multiplier m such that dividing by 2^k yields the quotient instead of the divisor n. That is,

$$\frac{m}{2^k} = \frac{1}{n}.$$

This method reduces dividing costs because the division is done by a power of two and allows computers to perform calculations via right-shifting.

Several other studies have applied large integer division to computers as well. For example, for the most basic algorithm, one proposed method calls for improving the calculation process of dividing three words by two words via a small-scale inverse that makes it more efficient [55]. In an evaluation in which it was applied to AMD Opteron and Intel Core 2 processors, it was found that this method could reduce the number of cycles by 15% and 40% for the former and latter processors, respectively, in comparison to conventional implementation results. Meanwhile, Harvey [56] focused on the middle product and attempted to improve the efficiency of the Katasuba multiplication applied in divide-and-conquer algorithms. More specifically, this process first considers the middle process in terms of polynomials and then, applies to integers. This study also proposed a method of treating carries as special and adding them to the calculation separately. The results obtained by applying this method and measuring the execution time showed performance improvements of more than 20% over GMP for calculations involving approximately 300 to 1,000 words.

Later, an improved version based on the above middle-product study, in which the divisor was truncated to a certain length in order to compute an approximation of a quotient, was proposed [57]. When this process is applied, the obtained quotient is used to calculate a middle product using the Karatsuba method, after which corrections for the quotient and middle product result are performed. The divide-and-conquer process used in that study is based on a slightly different method of dividing $2n$-1 words by n words. The execution time for $2n$-word / n-word with this algorithm on an AMD Opteron was approximately 15% to 20% faster when compared to GMP within $n = 966$.

Although some studies have evaluated performance levels in a domain-specific context (e.g., dividing a long integer by a single word [58] [59]), few studies have focused on using an arbitrary precision integer division to evaluate performance values, and no division studies that were evaluated using SIMD instructions have been identified, even though vectorization with SIMD instructions could potentially speed up computer arithmetic more than scalar instructions.

It should also be noted that many multiplication, addition, and subtraction operations are also performed inside large integer divisions, as will be discussed in this chapter. More specifically, since this research aims to speed up large integer division by vectorizing, a divide-and-conquer algorithm is implemented, in addition to naive division while targeting large integer sizes up to about 100,000 bits. Theoretically, the middle product method has the potential for use in fast Karatsuba multiplication, but from the vectorization point of view, there are many calculations that SIMD instructions cannot handle well and thus can have non-negligible impacts on performance levels. For example, it requires numerous branches as well as irregular multiplications and additions, which are not problems for scalar instructions. However, this middle product method has yet to be applied to GMP.

With the above points in mind, this chapter describes the implementation of a large integer division function that uses the normal Karatsuba method and vector instructions and then discusses a comparison of its performance in relation to GMP (scalar instructions).

5.2 Division Algorithm

In this study, a $n + m$-word dividend A and a n-word divisor B are represented as follows:

$$A = \sum_{i=0}^{n+m-1} a_i\beta^i, B = \sum_{j=0}^{n-1} b_j\beta^j,$$

where a_i and b_j are one word and β is 2^{64}. When considering a division A/B, the B is always assumed to be normalized. A normalized B means that B satisfies $\beta/2 \leq b_{n-1} < \beta$. If an input B is not normalized, A and B are multiplied by 2^k so that B is normalized. In other words, A and B are simply shifted to the left until the above condition is satisfied. Here, it should be noted that when the operands are left-shifted, the remainder R obtained from the calculation will ultimately be unnormalized (i.e., it will be shifted to the right by the same amount).

5.2.1 Basecase Division

Algorithm 6 shows the most basic division process [37]. In general terms, it divides two A words by one B word from the top of A. For each of these divisions, whether or not the calculated candidate quotient q_j is correct is determined. More specifically, it is determined whether or not the remainder $r = A - qB$ is negative. If it is negative, then the quotient is too large. In such cases, q_j is decremented by one, B is added to r, and these checks and adjustments are repeated until r becomes a positive value. At first glance, the cost of this `while` loop in line 11 appears to be high. However, since B is normalized, it takes just two loops, at most, to finish [50]. When computing r in an actual calculation, in-place subtractions for A can be performed.

To implement this algorithm straightforwardly, division instructions would normally be used. However, since division instructions are generally more expensive than addition, subtraction, and multiplication instructions, the proposed algorithms [55] provide a way to avoid them. Algorithm 7 shows how to divide three dividend words by two divisor words without using a division instruction. The inverse v used in this algorithm is obtained based on the Newton iteration expressed in the following equation:

$$x_{k+1} = x_k(2 - x_kd).$$

A particular feature of this algorithm is not only that it does not use division instructions, it avoids the costly full multiplication processes as much as possible and instead uses the

Algorithm 6 BasecaseDiv [37]

```
Input: $A = \sum_{i=0}^{n+m-1} a_i\beta^i, B = \sum_{j=0}^{n-1} b_j\beta^j, \beta/2 \leq b_{n-1} < \beta$
Output: $Q = \lfloor A/B \rfloor, R = A \bmod B$
 1: if $A \geq B\beta^m$ then
 2:     $q_m \leftarrow 1$
 3:     $A \leftarrow A - B\beta^m$
 4: else
 5:     $q_m \leftarrow 0$
 6: end if
 7: for $j$ from $m-1$ downto $0$ do
 8:     $q_j* \leftarrow \lfloor (a_{n+j}\beta + a_{n+j-1})/b_{n-1} \rfloor$
 9:     $q_j \leftarrow \min(q_j*, \beta - 1)$
10:     $A \leftarrow A - q_j B\beta^j$
11:     while $A < 0$ do
12:         $q_j \leftarrow q_j - 1$
13:         $A \leftarrow A + B\beta^j$
14:     end while
15: end for
16: return $Q = \sum_{k=0}^{m} q_k\beta^k, R = A$.
```

Algorithm 7 Div3by2 [55]

```
Input: $\langle u_2, u_1, u_0\rangle, \langle d_1, d_0\rangle, v$ with $\beta/2 \le d_1 < \beta$, $\langle u_2, u_1\rangle < \langle d_1, d_0\rangle$,
    $v = \lfloor(\beta^3 - 1)/\langle d_1, d_0\rangle\rfloor - \beta$
Output: $q_1 = \lfloor\langle u_2, u_1, u_0\rangle/\langle d_1, d_0\rangle\rfloor$, $\langle r_1, r_0\rangle = \langle u_2, u_1, u_0\rangle \bmod \langle d_1, d_0\rangle$
 1: $\langle q_1, q_0\rangle \leftarrow vu_2$
 2: $\langle q_1, q_0\rangle \leftarrow \langle q_1, q_0\rangle + \langle u_2, u_1\rangle$
 3: $r_1 \leftarrow (u_1 - q_1 d_1) \bmod \beta$
 4: $\langle t_1, t_0\rangle \leftarrow d_0 q_1$
 5: $\langle r_1, r_0\rangle \leftarrow (\langle r_1, u_0\rangle - \langle t_1, t_0\rangle - \langle d_1, d_0\rangle) \bmod \beta^2$
 6: $q_1 \leftarrow (q_1 + 1) \bmod \beta$
 7: if $r_1 \ge q_0$ then
 8:     $q_1 \leftarrow (q_1 - 1) \bmod \beta$
 9:     $\langle r_1, r_0\rangle \leftarrow (\langle r_1, r_0\rangle + \langle d_1, d_0\rangle) \bmod \beta^2$
10: end if
11: if $\langle r_1, r_0\rangle \ge \langle d_1, d_0\rangle$ then
12:     $q_1 \leftarrow (q_1 + 1)$
13:     $\langle r_1, r_0\rangle \leftarrow \langle r_1, r_0\rangle - \langle d_1, d_0\rangle$
14: end if
15: return $q_1, \langle r_1, r_0\rangle$.
```

lower half product to proceed with the calculation. In addition, adjustment of the quotient in Algorithm 6 has been made simpler so a quotient can now be obtained with an error of (at most) 1 in this context [60]. In this process, the `while` statement in Algorithm 6 has been removed and replaced with a maximum of two `if` statements. The second `if` statement is applied when the first `if` statement fails to predict the quotient. However, according to the study that proposed the Div3by2 algorithm, the possibility of hitting this `if` statement is very low.

This algorithm made it possible to proceed with the division more efficiently than is possible with the BasecaseDiv. However, it is still somewhat inefficient when considered from the perspective of vectorization using SIMD instructions because the propagation of a carry from the least significant word must be taken into consideration when using SIMD instructions. For example, carries have to be dealt with after each second and sixth line of Algorithm 7, which computes the candidate quotient q_1. When using AVX-512IFMA, if any word in the q_1 vector becomes 53 bits due to a carry, it cannot be calculated correctly with this instruction (e.g., $q_1 d_1$ in Line 3 of Algorithm 7). However, the carry process inhibits efficient computation and should be reduced as much as possible.

Here, when mathematically considering lines 3 to 5 of this Algorithm 7, since u_1, u_0, d_1, d_0 and q_1 are all positive values,

$$
\begin{aligned}
r_1 &= u_1 - q_1 d_1 - n\beta \quad (n \geq 0) && (5.1)\\
\langle t_1, t_0 \rangle &= d_0 q_1 && (5.2)\\
\langle r_1, r_0 \rangle &= \{((u_1 - q_1 d_1 - n\beta)\beta + u_0) \\
&\qquad - d_0 q_1 - (d_1\beta + d_0)\} - n'\beta^2 \quad (n' \geq 0) \\
&= (u_1 - (q_1 + 1)d_1)\beta + u_0 - (q_1 + 1)d_0 - (n + n')\beta^2 \\
&= (u_1\beta + u_0) - (d_1\beta + d_0)(q_1 + 1) - (n + n')\beta^2. && (5.3)
\end{aligned}
$$

The presence of $q_1 + 1$ in equation (5.3) means that the increment in Line 6 of Algorithm 7 can be performed in advance. However, when $q_1 = \beta - 1$, $(q_1 + 1) \bmod \beta$ becomes 0, but $q_1 + 1$ does not become 0 in equation (5.3). Thus, when $q_1 \neq \beta - 1$, the sixth line of Algorithm 7 can be moved to after the second line. Moreover, moving Line 6 also modifies the computation of $\langle r_1, r_0 \rangle$ which becomes:

$$\langle r_1, r_0 \rangle \leftarrow (\langle u_1, u_0 \rangle - \langle d_1, d_0 \rangle \times q_1) \bmod \beta^2.$$

The only exception to this is when $q_1 = \beta - 1$, which is quite unlikely to occur. At this time, $\langle d_1, d_0 \rangle \times q_1$ will become 0. On the other hand, since $\langle d_1, d_0 \rangle$ in the fifth line of Algorithm 7 is subtracted independently of the value of q_1, the case when $q_1 = \beta - 1$ must be considered as a special pattern. In this case,

$$
\begin{aligned}
r_1 &= u_1 - (\beta - 1)d_1 - n\beta \\
&= u_1 + d_1 - (n + d_1)\beta \\
\langle t_1, t_0 \rangle &= d_0(\beta - 1) \\
\langle r_1, r_0 \rangle &= \{((u_1 + d_1 - (n + d_1)\beta)\beta + u_0) \\
&\qquad - d_0(\beta - 1) - (d_1\beta + d_0)\} - n'\beta^2 \\
&= (u_1 - d_0)\beta + u_0 - (n + n' + d_1)\beta^2. && (5.4)
\end{aligned}
$$

Equation (5.4) means that when $q_1 = \beta - 1$, $\langle r_1, r_0 \rangle$ can be computed with only one subtraction. Therefore, the computation of $\langle r_1, r_0 \rangle$ at this time becomes as follows:

$$
\begin{aligned}
&r_1 \leftarrow (u_1 - d_0) \bmod \beta \\
&r_0 \leftarrow u_0.
\end{aligned}
$$

Based on these results, a modified version of the three-word / two-word division algorithm is proposed that is easier to process with SIMD instructions than Algorithm 7. The modified algorithm is given in Algorithm 8.

Algorithm 8 OurDiv3by2

```
Input: $\langle u_2, u_1, u_0\rangle, \langle d_1, d_0\rangle, v$ with $\beta/2 \le d_1 < \beta$, $\langle u_2, u_1\rangle < \langle d_1, d_0\rangle$,
    $v = \lfloor(\beta^3 - 1)/\langle d_1, d_0\rangle\rfloor - \beta$
Output: $q_1 = \lfloor\langle u_2, u_1, u_0\rangle/\langle d_1, d_0\rangle\rfloor$, $\langle r_1, r_0\rangle = \langle u_2, u_1, u_0\rangle \bmod \langle d_1, d_0\rangle$
 1: $\langle q_1, q_0\rangle \leftarrow vu_2$
 2: $\langle q_1, q_0\rangle \leftarrow \langle q_1, q_0\rangle + \langle u_2, u_1\rangle$
 3: $q_1 \leftarrow (q_1 + 1) \bmod \beta$
 4: if $q_1 = 0$ then
 5:     $r_1 \leftarrow (u_1 - d_0) \bmod \beta$
 6:     $r_0 \leftarrow u_0$
 7: else
 8:     $\langle r_1, r_0\rangle \leftarrow (\langle u_1, u_0\rangle - \langle d_1, d_0\rangle \times q_1) \bmod \beta^2$
 9: end if
10: if $r_1 \ge q_0$ then
11:     $q_1 \leftarrow (q_1 - 1) \bmod \beta$
12:     $\langle r_1, r_0\rangle \leftarrow (\langle r_1, r_0\rangle + \langle d_1, d_0\rangle) \bmod \beta^2$
13: end if
14: if $\langle r_1, r_0\rangle \ge \langle d_1, d_0\rangle$ then
15:     $q_1 \leftarrow (q_1 + 1)$
16:     $\langle r_1, r_0\rangle \leftarrow \langle r_1, r_0\rangle - \langle d_1, d_0\rangle$
17: end if
18: return $q_1, \langle r_1, r_0\rangle$.
```

One of the advantages of this method is that it sometimes makes it possible to calculate the quotient q with only one carry process. In the conventional method, the carry process of q must always be done twice (after the second line and before the 15th line of Algorithm 7). For the former, the carries are settled to correctly calculate the multiplication operation, such as $q_1 d_1$, and for the latter to return the final quotient due to the increment in line 6. The carry process should be avoided as much as possible when using SIMD instructions because of the data dependency requirement. On the other hand, in the proposed method, even though carries have to be dealt with for multiplication operations, if the `if` statements for adjusting q_1 are false, q_1 can be simply returned and the correct quotient can be obtained without dealing with carries.

Table 5.1: Values of OurDiv3by2 when q_1 overflows

line	variable	high	low
2	$\langle q_1, q_0 \rangle$	0xffffffffffff	0x1867e2b09a9ff
3	q_1		0x0
5	r_1		0x41c9b2a5ea8b7
6	r_0		0x09ceb67587638

Another advantage of this method is that it reduces the number of subtraction operations. More precisely, it replaces them with additions and fuses them into the multiplication operations. This not only makes the process simpler, it also makes it possible to utilize AVX-512IFMA more effectively. In actual processing, calculation of the upper half product of the d_1q_1 in $\langle d_1, d_0 \rangle \times q_1$ is omitted. Furthermore, even though it is true that the number of branches by the `if` statement increases by one compared to the original Div3by2, the merits of the proposed method outweigh this disadvantage. That is because AVX-512 has instructions that simultaneously compare eight words. Therefore, OurDiv3by2, which uses branches, maintains a higher data parallelism yield than the carry process. Furthermore, one branch is lighter than sequentially processing eight words of carries, resulting in faster processing in total. This algorithm, which is the core of the division calculations in this study, is called repeatedly when dividing large integers. Specifically, lines 8 to 14 of Algorithm 6 are replaced with the proposed algorithm and calculations are then performed. Improving the efficiency of this calculation contributes directly to speeding up large integer divisions.

5.2.2 Checking OurDiv3by2

In the previous section, a modified version of the original Div3by2 was proposed. GMP uses Algorithm 7, and lines 3 through 6 of this algorithm are modified. Therefore, specific values are used to determine if the calculation results of lines 3 through 9 of the proposed algorithm are the same as the original. If they match, then they are equivalent to GMP.

To accomplish this, a pattern where q_1 overflows (special pattern) is first examined, where $\langle u_2, u_1, u_0 \rangle =$ 0xe399b726e66e9796eedbb3d40209ceb67587638 and $\langle d_1, d_0 \rangle =$ 0xe399b726e66ea37a53b1552b4b. Tables 5.1 and 5.2 show the values that are assigned to each line of the algorithms. Since both algorithms are processed the same up to the second line, the description starts from the second line. Note that, when "high" is assigned, it refers

Table 5.2: Values of Div3by2 when q_1 overflows

line	variable	high	low
2	$\langle q_1, q_0 \rangle$	0xffffffffffff	0x1867e2b09a9ff
3	r_1		0x5d08a4e223aec
4	$\langle t_1, t_0 \rangle$	0x37a53b1552b4a	0xc85ac4eaad4b5
5	$\langle r_1, r_0 \rangle$	0x41c9b2a5ea8b7	0x09ceb67587638
6	q_1		0x0

Table 5.3: Values of OurDiv3by2 when q_1 does not overflow

line	variable	high	low
2	$\langle q_1, q_0 \rangle$	0x93514f3f7a3d5	0x8958987917788b
3	q_1		0x93514f3f7a3d6
8	$\langle r_1, r_0 \rangle$	0xb2b150afe8638	0x6220c6a7c9aae

Table 5.4: Values of Div3by2 when q_1 does not overflow

line	variable	high	low
2	$\langle q_1, q_0 \rangle$	0x93514f3f7a3d5	0x8958987917788b
3	r_1		0xba06918e68b68
4	$\langle t_1, t_0 \rangle$	0x494bacf000d4f	0x368b9a08adf83
5	$\langle r_1, r_0 \rangle$	0xb2b150afe8638	0x6220c6a7c9aae
6	q_1		0x93514f3f7a3d6

to the upper part of the two words. These results show that the final q_1, q_0, r_1, and r_0 values are consistent between the two algorithms.

Next, a case where q_1 does not overflow (which is the pattern in most cases) is examined, where $\langle u_2, u_1, u_0 \rangle =$ 0x6d5bdbcdd56bd1b291ebdb68c8180acdc339428 and $\langle d_1, d_0 \rangle =$ 0xbe0993ee7f7e07f5e6d12c19f7. Tables 5.3 and 5.4 show that, here as well, the final values are in agreement. The values in these tables are the same as those obtained by the actual GMP calculation method.

5.2.3 Divide-and-Conquer Division

In the previous section, the most basic division algorithms were discussed. However, when considering a $2n$-word / n-word division, the computational complexity of this algorithm is $O(n^2)$. This means that doubling n roughly quadruples the division execution time.

Divide-and-conquer division [52] algorithms are asymptotically faster than Basecase

Algorithm 9 RecursiveDiv [52]

```
Input: $A = \sum_{i=0}^{2n-1} a_i\beta^i, B = \sum_{j=0}^{n-1} b_j\beta^j, \beta/2 \le b_{n-1} < \beta$
Output: $Q = \lfloor A/B \rfloor, R = A \bmod B$
 1: $k \leftarrow \lfloor n/2 \rfloor$
 2: if $k$ is odd or $n \le$ DIV_LIMIT then
 3:     return $(Q, R) \leftarrow$ BasecaseDiv$(A, B)$
 4: end if
 5: $A_1 \leftarrow A/\beta^k, A_0 \leftarrow A \bmod \beta^k$
 6: $(Q_1, R') \leftarrow$ Div3by2Long$(A_1, B)$
 7: $A_0 \leftarrow R'\beta^k + A_0$
 8: $(Q_0, R) \leftarrow$ Div3by2Long$(A_0, B)$
 9: $Q \leftarrow Q_1\beta^k + Q_0$
10: return $(Q, R)$.
```

division algorithms. Algorithm 9 shows the divide-and-conquer division algorithm labeled RecursiveDiv, which is based on extending the word length in line 8 of Algorithm 6, where two words are divided by one. In other words, it is a method for recursively proceeding with the division by considering it as $2n$-word / n-word. When the number of words in the divisor becomes less than `DIV_LIMIT`, Basecase division is performed. In the main body of this algorithm, A is regarded as four parts. Specifically,

$$A = A_3\beta^{3k} + A_2\beta^{2k} + A_1\beta^k + A_0,$$

where $k = n/4$. First, the top three parts are divided by B, after which the remainders (R_1 and R_0) obtained from this calculation are combined with the remaining A_0 and divided by B as follows:

$$Q_0 = \frac{A_3\beta^{2k} + A_2\beta^k + A_1}{B},$$
$$Q_1 = \frac{R_1\beta^{2k} + R_0\beta^k + A_0}{B}.$$

These calculations are an extension of the calculation of dividing three words by two words.

Algorithm 10 shows the computation of $3n$-word / $2n$-word. This is also considered by

Algorithm 10 Div3by2Long [52]

```
Input: $A = \sum_{i=0}^{3n-1} a_i\beta^i, B = \sum_{j=0}^{2n-1} b_j\beta^j, \beta/2 \le b_{n-1} < \beta$
Output: $Q = \lfloor A/B \rfloor, R = A \bmod B$
 1: $A_1 \leftarrow A/\beta^n, A_0 \leftarrow A \bmod \beta^n$
 2: $B_1 \leftarrow B/\beta^n, B_0 \leftarrow B \bmod \beta^n$
 3: $(Q, R') \leftarrow$ RecursiveDiv$(A_1, B_1)$
 4: $A_0 \leftarrow R'\beta^n + A_0$
 5: $R \leftarrow A_0 - QB_0$
 6: if $R < 0$ then
 7:     $Q \leftarrow Q - 1$
 8:     $R \leftarrow R + B$
 9:     if $R < 0$ then
10:         $Q \leftarrow Q - 1$
11:         $R \leftarrow R + B$
12:     end if
13: end if
14: return $(Q, R)$.
```

splitting A in the same way as RecursiveDiv as follows:

$$\begin{aligned} A &= A_2\beta^{2n} + A_1\beta^n + A_0, \\ B &= B_1\beta^n + B_0, \\ Q &= \frac{A_2\beta^n + A_1}{B_1}. \end{aligned}$$

For quotient Q, since Algorithm 9 can be used to divide $2n$ words by n words, Recursive-Div is again called recursively. The overall computational complexity of the divide-and-conquer division depends on the internal use of the multiplication operation; specifically, $O(M(n)\text{log}(n))$, where $M(n)$ stands in for the complexity of an n-word multiplication algorithm. For a mid-range multiplication operation, the Karatsuba method [39] is used. When the operand size increases, the k-way Toom–Cook method [48] and the method based on a Fast Fourier Transform [61] multiplication are used. In this study, the Karatsuba method is applied to large integer multiplication operations.

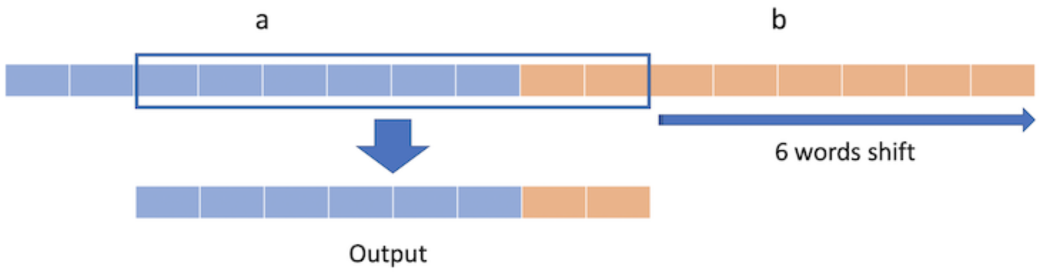


Figure 5.1: Overview of `valignq` process when passing two zmm registers (a and b) and an immediate value 6.

5.3 Implementation

5.3.1 AVX-512 Instructions for Implementation

In this section, instructions will be introduced that are particularly important for the implementation discussed in this research.

The first and most important instruction is AVX-512IFMA, which takes three operands in total and computes a 104-bit intermediate result by adding a 64-bit integer to a result of multiplying two 52-bit integers. It is necessary to use two instructions (`vpmadd52huq` and `vpmadd52luq`) to get the complete result of this calculation; the former to get the upper 52 bits of the 104 bits, and the latter to get the lower 52 bits [19]. As a result, a numerical expression with a radix of 2^{52} is required to perform calculations using this instruction set. However, since multiplying large integers involves numerous multiplication and addition operations, it is more convenient to calculate them in the reduced-radix representation. In addition, since large integer multiplication works well with AVX-512IFMA, which can perform addition and multiplication operations simultaneously, these instructions were used in many of the implementations in this research. The second is `valignq`, which takes two zmm registers and one immediate value [19]. Figure 5.1 shows an example process of `valignq`. This instruction first concatenates the two zmm registers to obtain an intermediate result of 1,024 bits, and then right-shifts this result by $64n$ ($0 \leq n \leq 7$) bits. Finally, it stores the lower 512 bits of the shifted result in a zmm register. In other words, it can shift the data by word units. A `valignq` instruction makes it possible to adjust the position for additions of intermediate results within a large integer multiplication or for the addition of carries. The third is the comparison instruction `vpcmpuq` [19]. This instruction performs 64-bit unit comparisons in parallel and stores the results in a mask register. If the result is true, a bit is set to the corresponding position. At the assembly level, the specific behavior is determined by the fourth immediate value argument. The following eight types can be specified: equal (EQ), less than (LT), less than or equal (LE), false (FALSE), not equal (NEQ), greater than or equal (NLT), greater than (NLE), or true (TRUE).

In terms of implementation, the following intrinsic functions were used to implement the instructions introduced above. `__m512i` is a type that indicates an integer 512-bit zmm register.

- `_mm512_madd52hi_epu64 (__m512i a, __m512i b, __m512i c)`
- `_mm512_madd52lo_epu64 (__m512i a, __m512i b, __m512i c)`
- `_mm512_alignr_epi64 (__m512i a, __m512i b, const int imm8)`
- `_mm512_cmpeq_epu64_mask (__m512i a, __m512i b)`

As for the compare instruction, the above “`cmpeq`” function shows one of the eight, which indicates equal. While at the assembly level it needs to be specified by an immediate value, which can be determined at the C/C++ level by its function name.

5.3.2 Basecase Division

Large integers are represented in 64-bit unsigned long arrays. Since AVX-512IFMA takes 52-bit operands for multiplication, large integers should be converted into the reduced-radix representation of that length per word before starting a division. Moreover, regardless of whether the target algorithm is a Basecase or divide-and-conquer division, a divisor B must be normalized as a prerequisite. Therefore, A and B are adjusted so that B is normalized in a 52-bit representation. To accomplish this, the number of missing bits in the most significant word b_{n-1} is precomputed in order to satisfy the $\beta/2 \leq b_{n-1} < \beta$ condition when large integers are converted from a 64-bit to a 52-bit representation. Next, A and B are entirely shifted to the left by the value obtained from that precomputation, and then convert to the reduced-radix representation. After the division, the remainder is shifted to the right by the same amount because it needs to be unnormalized, and then convert both the quotient and remainder back to the normal 64-bit representation.

Focusing on the BasecaseDiv algorithm, the inverse of the divisor needs to be calculated in advance. In this study, since $\beta = 2^{52\times8}$ in Div3by2 is considered, the reciprocal is calculated for 416 bits, which is represented as a fixed-point number. To obtain this value, the Newton–Raphson method is used. Since it is known that at least 416 bits are sufficient for the Div3by2 algorithm, it can be implemented specifically. Furthermore, since a zmm register is 512 bits wide, the calculation can be performed using only zmm registers, and associated memory access costs can be circumvented.

The cores of the BasecaseDiv algorithm are the Div3by2 and in-place subtractions to A in the main loop. The Div3by2 algorithm handles the selection of the candidate quotient q_j and its adjustment. For the q_jB computation, since it is known that the q_j length is eight words, a fixed-length module can be implemented and called repeatedly in order to make its computation more efficient. In this module, eight words are multiplied by eight words. More specifically, one zmm register is multiplied by another, which makes it easy to store the results in some registers during the calculation.

It also allows limitation of memory accesses for loading operands and storing the final result. Furthermore, since it is also known that the q_j determined by the Div3by2 is used repeatedly in this multiplication operation, passing the zmm register storing q_j directly to that module allows for more memory access reduction. During multiplication and subtraction operations in $A - q_jB$, since carries and borrows are accumulated due to the reduced-radix representation, the calculations can be performed with SIMD instructions without worrying about those processes. When the subtractions are finally completed, the carries are dealt with for the first time, and the process moves to the next loop.

5.3.3 Our Div3by2

Next, the core OurDiv3by2 algorithm is focused on. The operands u_i, d_i, and v each have 416-bit precision (52bits $\times$ 8 words). That is, as a function, it divides 1,248 bits by 832 bits. In this function, multiplication is done twice (in line 1 and line 8) in Algorithm 8. In both cases, since the operand lengths are fixed, specialization can be implemented. This allows the calculation results to be accumulated in zmm registers, as described above. In addition, since it is a small-scale multiplication, all the calculations were unrolled to make the process more efficient. The only other arithmetic operations are addition and subtraction, which can also store results in registers without memory access. Thus, essentially, this

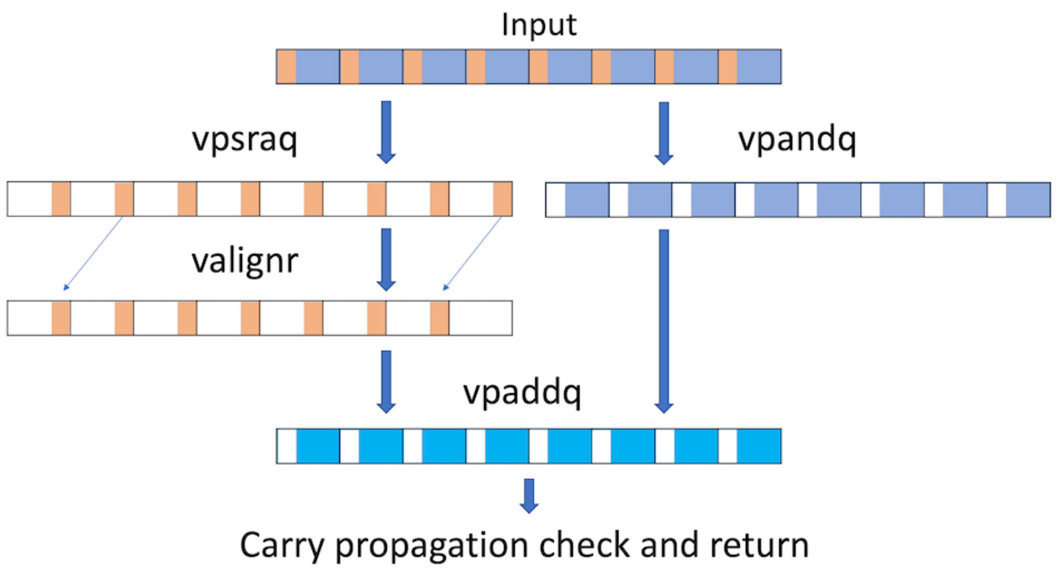


Figure 5.2: Overview of carry process with AVX-512 using an optimistic approach.

function only accesses memory when loading arguments and storing calculation results. However, exceptions may occur when calculations are implemented using SIMD instructions. More specifically, when carries are processed from the least significant word, carry propagation may occur, thereby resulting in data dependency that makes vectorization difficult. Fortunately, however, such occurrences are unlikely because they would mean that their neighboring words already have values of almost $2^{52} - 1$. Therefore, an optimistic approach (see below for details) was adopted and the carry process was implemented with AVX-512 instructions for OurDiv3by2.

Figure 5.2 shows an overview of the carry process. The blue and orange areas represent 52-bit-wide data and a 12-bit-wide carry, respectively. The white areas in the registers contain zeros. This approach simultaneously acquires the carries of all eight words stored in a zmm register and adds them to the neighboring words. The carry acquisition itself is very simple, using the `vpsraq` instruction to shift 52 bits to the right. Next, the AND operation `vpandq` is passed at $2^{52} - 1$ to clear the carries from the original data. To add the carries to the next word, the `valignr` instruction shifts them to the left by one word and then adds them using `vpaddq`. Finally, the next process checks to see if there are any carries left. If any remain, it means that the value is greater than $2^{52} - 1$. The `vpcmpuq` instruction can then be used to detect values greater than $2^{52} - 1$ in the zmm register. If the output is not zero (even though the probability is very low), scalar instructions are used to deal with the propagating carries, which are processed sequentially in memory. After sequential processing, the memory output is loaded back into the zmm register and the next process is performed. In this way, eight carry words can essentially be calculated in most cases using five instructions. Moreover, since it can be done entirely in registers, it is faster than scalar instructions.

The comparisons of the values in lines 10 and 14 of Algorithm 8 can also be completed in the registers. In this Div3by2 context, a comparison of the two most significant words is sufficient in most cases. Therefore, it is usually completed by determining whether or not the most significant bit of the output of the `vpcmpuq` is set with the NLT flag. However, since there are some cases where the first words are the same, and the lower words are different, a comparison instruction is executed with the EQ flag at the same time. The following procedure is performed to check the most significant bits of the two outputs of EQ and NLE (for example, `eq` and `gt`, respectively) as the target bit to determine if $A \geq B$:

1. If `eq` is 0xff, it means $A = B$ and True is returned.
2. If the target bit of `eq` is 0, the same position of `gt` is checked, and returns True if 1, False if 0.
3. Otherwise, the target bit is changed to one lower and the procedure returns to step 2.

The third step is rarely executed. For the two-word comparison in line 14 of Algorithm 8, the program first checks if $r_1 = d_1$ by comparing the EQ flags. If this is the case, the above procedure is run for r_0 and d_0. If not, r_1 and d_1 are targeted. However, the former pattern seldom occurs.

5.3.4 Divide-and-Conquer Division

With some optimizations, it is possible to efficiently perform computations with SIMD instructions in the Basecase division algorithm. However, as the operand size increases, the execution time increases due to computational costs. Therefore, the divide-and-conquer algorithm is also implemented for larger sizes. Basically, this algorithm is applied when the size of B is larger than 320 words (i.e., 52 bits $\times$ 320 words = 16,640 bits) and set this threshold. However, when the length of A is relatively short compared to B (e.g.,

$1.5n$-word / n-word), the length of the quotient is correspondingly short, and the Basecase division process can process it faster within the range covered in this study. Therefore, in such cases, the Basecase algorithm is used to perform a division even if it exceeds the threshold.

There are two reasons for setting the threshold (`DIV_LIMIT` in Algorithm 9) at 320. The first is because it is computationally convenient. The decision to perform recursive processing depends on the divisor size. The maximum size targeted in this research is 1,024 words in a 64-bit representation, which would be 1,261 words if converted to a 52-bit representation. This number is close to 1,280 (= 320 $\times$ 4), and since 1,280 (or more specifically, 320) is a multiple of 64, it is convenient for optimization in terms of splitting and memory alignment.

Second, from the perspective of memory access latency, the Basecase division function employs in-place arithmetic for efficient computation. This function is also used in divide-and-conquer, which requires recursive processing to proceed so that the original data is not overwritten. Therefore, when calculating the Basecase in divide-and-conquer, the target values are copied to other memory areas in advance before calculation. For example, taking the first reason into consideration, if `DIV_LIMIT` = 160, the calculation is finally divided into eight parts when the divisor is 1,024 words in a 64-bit representation. However, this also increases the number of memory allocation, release, and copy operations, which increases latency in non-essential operations. For these two reasons, `DIV_LIMIT` = 320 was adopted in this study.

The first job of the divide-and-conquer algorithm is to calculate $2n$-word / n-word. However, a general division is not always applicable in this form. Furthermore, even though this method works well when the length of A is less than or equal to twice the length of B, issues can occur that make division difficult, such as when dividing $2n$+1 words by n words. Moreover, even if the lengths of the A and B inputs are $2n$ and n, respectively, they may change to the form $2n$+1-word / n-word as a result of the left shift of A and B after normalizing. Therefore, it is necessary to be flexible when dealing with this issue in order to correctly calculate using divide-and-conquer division. To resolve this, the apparent size of A and B are extended to be $2n$ and n using zeros. If the most significant word side is filled with zeros, since the condition that B is normalized is not satisfied, the least significant word side is filled. Let the original word lengths of A and B be a and b, respectively, and the scaling number be s. When $a > 2b$, since if $2(b + s) \geq a + s$, then it becomes computable by divide-and-conquer,

$$\begin{aligned}2(b+s) &\geq a+s,\\ s &\geq a-2b.\end{aligned}$$

Here, A and B should be shifted to the left by at least $a-2b$ words. Furthermore, from the perspective of using SIMD instructions, it is more convenient if each operand length remains a multiple of the vector length (eight words in the case of AVX-512) even if the operands are split by recursion. Therefore, taking into consideration the number of recursions, scaling s, and vector length, the final length of A is adjusted, and A and B are then shifted to the left so that A does not exceed twice the length of B.

For certain dividend lengths, this scaling method can be used alone to perform the divide-and-conquer process at high speed. However, for example, for $3n$-word / n-word, the above scaling results in a calculation of $4n$-word / $2n$-word. This means that the divisor size is doubled, and there are numerous non-essential calculations. In this case, even if the algorithm is asymptotically fast, the effective performance level will be low. One way to deal with this issue is to treat the calculation as a large Basecase division and apply the divide-and-conquer process in each Basecase loop [52]. That is, taking $3n$-word / n-word as an example, $2n$ words are divided by the n word twice from the top. In this way, when the A length becomes somewhat long relative to B, it can efficiently be calculated by calling the divide-and-conquer division multiple times based on the Basecase method.

Next, RecursiveDiv and Div3by2Long are called alternately until the length of the divided block falls below the threshold, and then let them recurse. Although the condition in the second line of Algorithm 9 that includes the size k is odd, since k is adjusted so that it is always even by taking into consideration the aforementioned vector length, only the threshold (320 words as mentioned above) is used in the implementation. Furthermore, even though lines 7 and 9 of Algorithm 9 and line 4 of Algorithm 10 all appear to be performing an addition process, each addition operand is, in fact, placed in an adjacent memory location and is independent of the others. Therefore, the processes are actually completed by calculating in-place for R' and Q. Here, it should be noted that the QB_0 calculation in line 5 of Algorithm 10 is a multiple precision integer multiplication. It is also known that the multiplier and multiplicand are both the same size and large enough to permit the application of the divide-and-conquer method. Therefore, the Karatsuba method is used to compute them, which is asymptotically faster than naive multiplication [37].

5.3.5 Implementation Summary

Our implementation for large integer division was described in sections 5.3.2 through 5.3.4. This section provides a summary of the three division methods implemented in this study.

1. **Basecase**
   The core part is OurDiv3by2, which divides 24 words by 16 words in a 52-bit representation. Since the size of the OurDiv3by2 computation is known, the design is such that, to the greatest extent possible, the computation can be done using only registers.

2. **Divide-and-conquer with scaling technique**
   The usual divide-and-conquer algorithm, which generally performs $2n$ / n calculations. If the size of an input dividend is larger than twice the size of a divisor, it is adjusted to perform an apparent $2n$ / n calculations by using zeros to expand the size of both values beforehand.

3. **Basecase-based divide-and-conquer**
   A method in which lines 8-14 of Algorithm 6 are replaced by divide-and-conquer. This method is used to avoid increasing non-essential computations when the size of the divisor and the dividend become too large as a result of applying the divide-and-conquer process with the scaling technique.

5.4 Evaluation

5.4.1 Evaluation Environment

For evaluations, a host machine equipped with an Intel Core i3-8121U (Cannon Lake microarchitecture) was used, and all experimental programs were implemented in the C language and compiled through an Intel oneAPI DPC++ Compiler `icx` version 2021.2.0. The evaluation comparison target was the GNU Multiple Precision Arithmetic Library (GMP) [15] version 6.2.1, which was compiled by the same compiler. A single core and a single thread were used to run and evaluate the programs and a system time seeded `rand` function was used to generate the divisor and dividend. To evaluate performance levels in terms of execution time, the target functions were executed 5,000 times and obtained the average times.

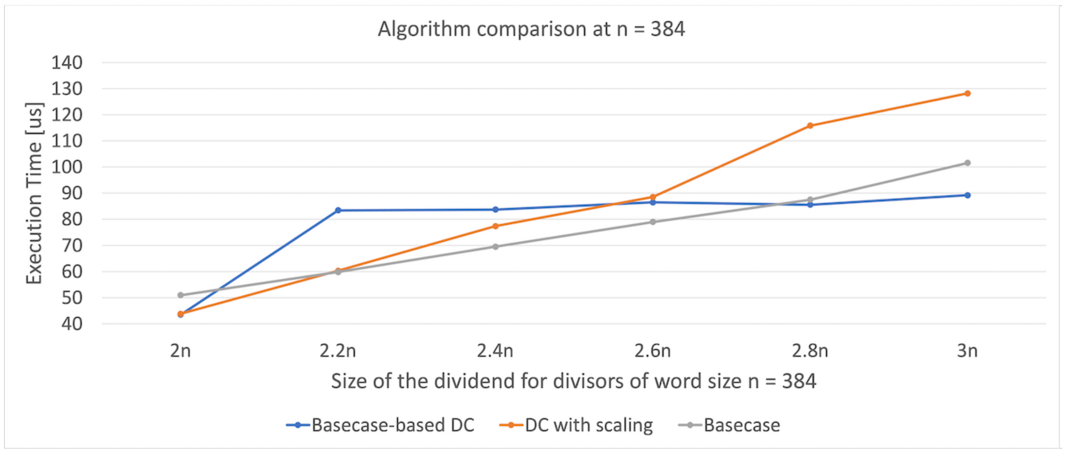


Figure 5.3: Execution times associated with the Basecase and divide-and-conquer methods when $n = 384$.

5.4.2 Comparing Algorithms

The algorithms that were implemented in section 5.3 were applied to the division function of this study. As a basic premise, as mentioned in subsection 5.3.4, the algorithms are switched between the Basecase and divide-and-conquer processes based on 320 words in a 52-bit representation for the divisor word length B. Next, since two divide-and-conquer methods (scaling and Basecase-based) were used, rough estimates of the size at which the algorithms would be switched are made.

Figures 5.3, 5.4, and 5.5 show execution time comparisons for the three methods when the input divisor size is set to 384, 512, and 640 words, respectively. The vertical axis is the execution time (us), and the horizontal axis is the dividend size A (e.g., $2n$ implies 1,024 words in 64-bit representation in Figure 5.4). *Basecase-based DC* shows the Basecase algorithm with lines 8 - 14 in Algorithm 6 replaced with the divide-and-conquer process, while *DC with scaling* shows the divide-and-conquer algorithm using the scaling technique. The reason why the blue line is almost constant from 2.2n to $3n$ is that the $2n$-word / n-word divide-and-conquer division is performed twice. In other words, $3n$-word / n-word is calculated in all of these ranges.

From these graphs, it can be seen that when the size of A is equal to or less than 2.4n, the scaling method is fastest in Figures 5.4 and 5.5, which means this method is effective for some large divisions. However, for larger sizes, the proportion of non-essential calculations increases as the scale increases further. As a result, it is by far the slowest of the three

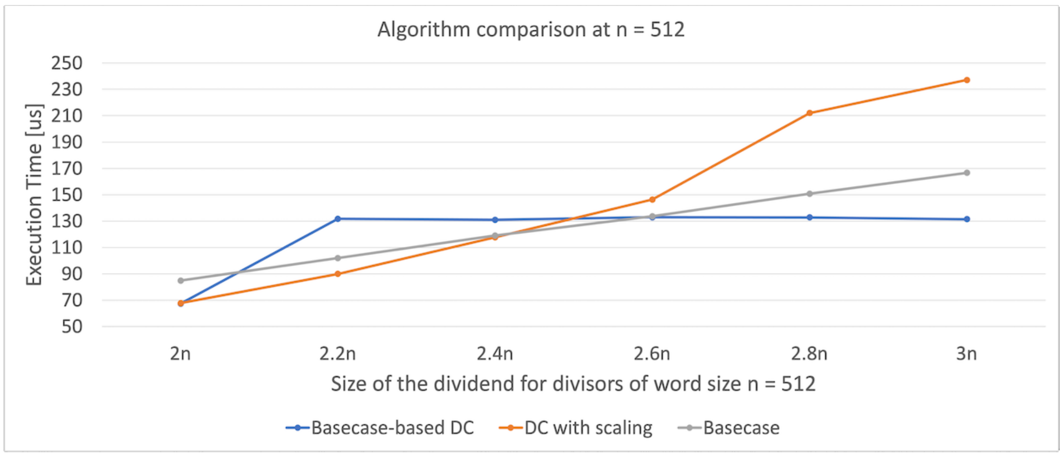


Figure 5.4: Execution times associated with the Basecase and divide-and-conquer methods when $n = 512$.

methods in that range. It can also be seen that when $n \geq 512$, the other divide-and-conquer method is fast enough to compensate for it and that this method is the fastest, especially for sizes 2.6n and above. It should also be noted that when the size of A is around 2.5n, the execution times for all three methods are almost the same, and while the pure Basecase method tends to be the fastest, the differences are just a few microseconds. On the other hand, at $n = 384$, even though the Basecase method is faster in the middle range, there are some areas where the divide-and-conquer process is faster. In other words, the larger the value of n, the more effective the divide-and-conquer method is in speeding up the process. Based on the above results and for the sake of simplicity, the divide-and-conquer method was used when the size of A became 2.5 times larger than B. Strictly speaking, although the Basecase method is faster around this point, from the viewpoint of arbitrary precision division, the divide-and-conquer process can be calculated more efficiently as the size of B increases. At the same time, however, these results show that even the Basecase algorithm is fast enough for certain sizes, especially in Figure 5.3.

5.4.3 GNU MP Comparisons

For the comparisons discussed in this section, GMP's `mpz_fdiv_qr` was used, which is an arbitrary precision division function that computes both a quotient and remainder. It should be noted that the implemented division function includes conversion from a 64-bit to a 52-bit representation, normalization (including the scaling process described above),

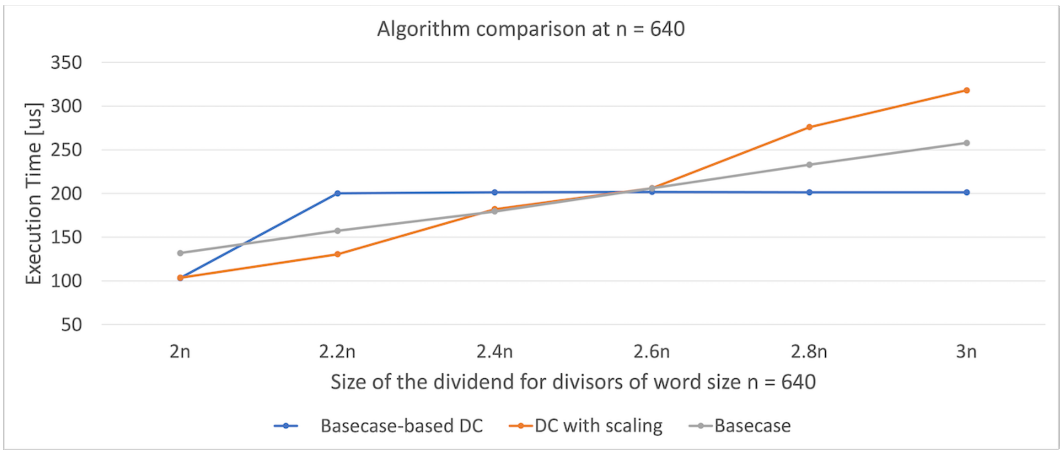


Figure 5.5: Execution times associated with the Basecase and divide-and-conquer methods when $n = 640$.

Table 5.5: Execution times for 1.5n-word / n-word division.

n	Proposed (Full)	Proposed (Kernel)	Ratio	GMP	Ratio
32	1.11	1.05	0.94	0.67	1.65
64	2.04	1.93	0.94	1.99	1.02
128	4.37	3.85	0.88	6.07	0.72
256	13.33	12.13	0.91	18.43	0.72
512	45.60	42.35	0.93	54.84	0.83
1024	164.67	157.40	0.95	158.88	1.03

division kernel, unnormalization, and reversion back to a 64-bit representation; all of which are included in the measurement. For large integers, an upper limit of 2,560 words (i.e., 163,840 bits) was set for the length of the dividend A passed to the implemented functions. Since the implementation targets arbitrary-precision division operations, measurements for various sizes were conducted. More specifically, 1.5n-word, 2n-word, 2.5n-word, 3n-word, and 4n-word were divided by n-word, where n is the word length of the divisor B.

Table 5.5 shows execution time comparisons when the A length is 1.5 times the B length. *Proposed (Full)*, *Proposed (Kernel)*, and *GMP* represent the execution time (us) for the entire division process, kernel processing (i.e., without the AVX-512-specific process of 52-bit and 64-bit conversion), and GMP, respectively. *Ratio* indicates the ratio to *Proposed (Full)*. Here, it can be seen that, for smaller sizes, GMP was clearly faster at n = 32 and

Table 5.6: Execution times for $2n$-word / n-word division.

n	Proposed (Full)	Proposed (Kernel)	Ratio	GMP	Ratio
32	1.36	1.25	0.92	1.26	1.08
64	2.76	2.60	0.94	3.87	0.71
128	7.58	7.04	0.93	12.07	0.63
256	24.15	23.30	0.96	36.17	0.67
512	67.59	64.24	0.95	109.53	0.62
1024	208.55	199.22	0.96	316.47	0.66

Table 5.7: Execution times for $2.5n$-word / n-word division.

n	Proposed (Full)	Proposed (Kernel)	Ratio	GMP	Ratio
32	1.68	1.62	0.97	1.86	0.90
64	3.71	3.47	0.94	5.63	0.66
128	10.77	10.06	0.93	17.68	0.61
256	35.52	33.42	0.94	54.37	0.65
512	131.03	124.81	0.95	163.53	0.80
1024	409.45	395.05	0.96	472.23	0.87

2% faster at n = 64. This is because the calculation scale is too small for the division with AVX-512 and due to the specific pre- and post-processing required to use the AVX-512 instructions (more specifically, the time for conversion to and from a 52-bit representation), and the reciprocal number calculation required for Algorithm 8. While GMP also calculates a 64-bit reciprocal, a 416-bit reciprocal is calculated. Therefore, the proposed calculation cost is higher than that for GMP. GMP also had a slight advantage over the proposed method for 1,024 words due to the differences in the algorithms used, primarily because GMP uses the divide-and-conquer algorithm for its division while Basecase division is used due to the reasons mentioned above. However, even though the most basic algorithm is used, AVX-512 instructions allow calculations to be performed at speeds closer to GMP, which uses an asymptotically faster algorithm. Thus, this result shows that the AVX-512 instructions can offer a performance improvement over scalar instructions in multiple-length integer division. In fact, with a performance improvement of up to about 30%, the implementation is notably faster in the middle range.

Tables 5.6 and 5.7 show execution time comparisons when the A size is set to $2n$ and $2.5n$, respectively. In these comparisons, the divide-and-conquer algorithm is used when the word lengths n of B are 512 and 1,024. Focusing on n = 32, it can be seen that the

Table 5.8: Execution times for $3n$-word / n-word division.

n	Proposed (Full)	Proposed (Kernel)	Ratio	GMP	Ratio
32	1.94	1.86	0.96	2.44	0.80
64	4.80	4.45	0.93	7.54	0.64
128	13.80	13.04	0.94	23.68	0.58
256	46.83	44.76	0.96	71.92	0.65
512	132.22	127.77	0.97	217.00	0.61

Table 5.9: Execution times for $4n$-word / n-word division.

n	Proposed (Full)	Proposed (Kernel)	Ratio	GMP	Ratio
32	2.52	2.44	0.97	3.49	0.72
64	6.78	6.12	0.90	11.03	0.61
128	20.04	18.76	0.94	35.24	0.57
256	74.32	69.20	0.93	106.96	0.69
512	232.28	204.48	0.88	324.99	0.71

difference in execution time with GMP has declined compared to the time in Table 5.5. However, when the size of A is $2.5n$, the situation is reversed, and the proposed method is superior to the library. For larger sizes, when n is equal to or greater than 512, the implementation is faster than GMP due to the divide-and-conquer algorithm. Overall, the average performance improvement is about 25% over GMP. Here, a comparison between the Basecase and divide-and-conquer methods is performed. When the size of A is $1.5n$, if the divide-and-conquer process is applied when n = 1,024, $2n$-word / n-word calculations are necessary due to the nature of the algorithm. Based on this, it can be seen that the Basecase algorithm is about 40 microseconds faster than n = 1,024 in Table 5.6. Therefore, as described in subsection 5.3.4, when the length of A is not excessive, the Basecase method is more efficient.

Finally, the effect of a larger dividend is evaluated. Tables 5.8 and 5.9 show execution time comparisons when the dividend is three and four times larger than the divisor, respectively. In both cases, the implemented program is faster than GMP for all sizes. Throughout, it is found that the percentage of execution time for the 64-bit to 52-bit conversion, which is a process unique to the AVX-512, is generally about 5 to 10% of the total. The $1.5n$-word / n-word comparison shows that GMP is faster than the proposed approach, even with AVX-512, because it is still processing the most basic division at n = 1,024. Later, however, since it can be seen that the implementation is faster in all comparisons at

n = 512 and 1,024 using divide-and-conquer, the algorithmic performance improvements are considered to be confirmed.

To summarize, the proposed implementation resulted in average performance improvements of about 35%, which are higher than the case for $2n$ and $2.5n$. Furthermore, since GMP does not use SIMD instructions to calculate divisions, SIMD instructions are found to be more advantageous than scalar instructions, especially when the A size is large.

5.5 Conclusion

This paper reports on efforts to speed up large integer division by utilizing AVX-512 SIMD instructions. To accomplish this, asymptotically fast divide-and-conquer algorithms were incorporated into the most basic methods. Additionally, since the target is arbitrary precision divisions, two divide-and-conquer methods (Basecase-based and scaling) were implemented in an effort to meet this target. Furthermore, since this algorithm includes large integer multiplications, Karatsuba multiplication was applied to speed up the process.

To compute this multiplication more efficiently and thus take full advantage of the AVX-512IFMA instruction set, a divisor and dividend were converted into a reduced-radix representation of 52 bits per word. The Basecase division operation was also devised, together with a procedure that is friendlier to SIMD instructions than existing methods. This allowed implementation of a division function that combines some optimization techniques. We then evaluated the implemented program on a Cannon Lake microarchitecture processor and compared the execution times for various sizes with the GNU MP library.

In comparison with GMP, we found that this library was faster when the word length of the divisor was 32 (i.e., 2,048 bits) and the dividend was shorter than about 4,096 bits. For such relatively small divisions, GMP is currently superior because the cost of the processing that must be performed separately from the division kernel in order to use SIMD instructions is non-negligible. In a similar context, due to the difference in the algorithm used, GMP was faster for 1,536-word / 1,024-word (i.e., 98,304 bits divided by 65,536 bits) calculations. However, the difference in execution time was less than 5% even though GMP uses the divide-and-conquer algorithm while the proposed program uses the Basecase algorithm. For the other patterns, the implementation was faster, with average performance improvements of 25% to 35%. These results suggest that SIMD instructions are useful for speeding up the division of large integers.

Chapter 6

Conclusion

This thesis provided the results of applying SIMD instructions to large integer multiplication and division. Here, the results are summarized and future work is discussed.

6.1 Summary

The objective of this thesis was to improve the performance of large integer multiplication and division by using SIMD instructions in CPUs. To achieve this objective, implementation methods and algorithms that incorporate the reduced-radix representation and enable efficient computation using SIMD instructions were proposed. This minimized the effect of handling carries, which is a challenge in large integer arithmetic.

For large integer multiplication with Intel processors, AVX-512F and AVX-512IFMA multiplication instructions were implemented. The multiplier and multiplicand were converted into 2^{28}-radix and 2^{52}-radix representations, respectively. In terms of algorithms, the most basic multiplication method and the Karatsuba method were applied. Further optimizations were made by taking into account memory access and stall effects in the implementation. In addition to the multiplication kernel, SIMD instructions were used in the reduced-radix representation conversion process and AVX-512BW and AVX-512VBMI were utilized. For the performance evaluation, the implemented program was run on Xeon Phi Knights Landing and Cannon Lake processors and compared with GMP. The results showed that both the 32-bit AVX-512F and 52-bit AVX-512IFMA multiplications were faster than GMP on the target processors for operand sizes of 2,048 bits or more. For the Knights Landing processor, a performance gain of up to approximately 2.5x over GMP was obtained with AVX-512F instructions. For the Cannon Lake processor, a performance gain of approximately 2.97x was obtained with AVX-512IFMA.

Large integer multiplication was also examined for an Arm processor. It was performed on an A64FX processor capable of executing 512-bit SVE instructions. Since SVE has no special restrictions on the number of bits, unlike AVX-512IFMA, each operand was calculated by converting it to a 2^{56}-radix representation. However, the partial product must be consistently in a 2^{56}-radix reduced-radix representation during the calculation of the multiplication, which requires additional operations for SVE with 64-bit multiplication instructions. Therefore, a multiplication algorithm that maintains this consistency was proposed and applied to the implemented program. Since this method uses basic instructions such as shift and AND instructions, it could be implemented for architectures other than Arm. The program was compiled in two modes: one the Fujitsu compiler-based trad mode and the other the Clang/LLVM-based clang mode. In the performance evaluation, GMP was slightly faster than the implemented program in 2,048-bit multiplication due to latency and the increased number of instructions in the proposed algorithm; nevertheless, the difference was small. However, for larger operand sizes, the SVE implementation was faster. The performance gains were up to 36% with trad mode compilation and up to 31% with clang mode compilation.

AVX-512 was also applied to large integer division. AVX-512IFMA was used in this calculation because some conventional methods have algorithms that calculate the quotient by multiplying inverses and previous research has shown that AVX-512IFMA is particularly effective in speeding up multiplication. The inverse is calculated only once and the cost of this calculation is relatively small. Therefore, for an inverse with a larger number of digits, a larger number of candidate quotient digits can be calculated at once by the module, making this method a good match for SIMD instructions. A SIMD-instruction-friendly division algorithm was then proposed. It reduces the processing of the carry compared with that for an implementation based on the conventional method. In addition, a more optimal implementation was made for the division calculation. This is because, in the algorithm used in this study, carry propagation is much less likely to occur, so taking an optimistic approach to speculative parallel carry processing is not a problem in most cases. If carry propagation occurs as a result of a calculation using this approach, then sequential processing with scalar instructions is performed. The most basic division method and the divide-and-conquer method were used to implement this. The design is such that the operand size can be properly calculated by adjusting it in advance to match the preconditions in the divide-and-conquer method. The performance evaluation results show that the basic algorithm is faster when the dividend size is less than approximately 2.5 times the divisor size and that the divide-and-conquer method is faster for a large operand size. GMP was slightly faster for $2n$-word / n-word division at 2,048 bits, but the proposed implementation was faster for larger operand sizes. Furthermore, when the dividend size was large ($2.5n$, $3n$, and $4n$), the proposed method was faster even at 2,048 bits. On average, performance gains of 25% to 35% were obtained.

In summary, large integer multiplication and division computations can be accelerated by SIMD instructions, which are faster than GMP in all cases for large operand sizes. Thus, SIMD instructions and the reduced-radix representation speed up computation and thus the objective of this study has been achieved. SIMD instructions have long been included in CPUs, allowing bit-wide instructions such as AVX-512 and SVE to be executed. Starting with Intel processors, 512-bit-wide SIMD instructions are becoming more common in desktop processors. In the near future, these instructions will be available in many general-purpose computers. This thesis showed that various large integer arithmetic operations can be processed faster using these computers.

6.2 Future Work

This section gives suggestions for future work.

The most basic method and the Karatsuba method were used in the algorithm for large integer multiplication. However, there are various multiplication algorithms, such as the Toom–Cook method [48], that are known to be asymptotically faster for large integer multiplication [37]. The Toom–Cook method reduces the burden of multiplication processing by dividing one operand into three or more parts (in the Karatsuba method, one operand is divided into two parts). Implementing asymptotically faster algorithms with SIMD instructions should further speed up large integer multiplication, especially for operands larger than those considered in this thesis. Thus, in the future, these algorithms will be implemented with SIMD instructions and compared with the Karatsuba method. Furthermore, GMP performs multiplication by increasing the number of divisions to 3-way and 4-way forms when the size of the multiplier and multiplicand exceeds some threshold. Therefore, in future research, it will be desirable to implement multi-precision integer multiplication with the n-way form of the Toom–Cook method using SIMD instructions and investigate at which operand sizes the algorithm should be switched. This will produce behavior closer to that of GMP and will bring us closer to achieving arbitrary-precision arithmetic library with SIMD instructions.

The SVE2 instruction set has been introduced [49]. One of the instructions in this set targets large integer arithmetic. For example, this instruction set includes instructions that perform addition with a carry (`ADCLB` and `ADCLT`). These instructions may solve the carry problem that arises when SIMD instructions are used. Furthermore, once this problem is solved, it will be possible to perform large integer multiplication without the reduced-radix representation, thus eliminating the need for conversion processing. However, this SVE2 instruction is designed to store the carry in an adjacent word instead of the carry flag for the scalar instructions. This means that the number of words that can actually be added is halved. Therefore, even if these SVE2 instructions enable the computation of large integer multiplication without the reduced-radix representation, it does not necessarily mean that they can be processed faster than the method in this thesis, which uses this representation. A method that combined SVE and SVE2 instructions will be implemented and its performance will be compared with that of a method that uses only SVE and GMP.

In addition, the A64FX processor can execute 512-bit-wide SIMD instructions, whereas SVE instructions can be up to 2,048 bits. A larger bit width should allow for faster calculations, especially when the multiplier and multiplicand sizes are large. Therefore, the performance of the proposed approach will be evaluated on processors that can execute instructions with larger bit widths.

There are also several ways to speed up the division process. The first is to improve the efficiency of the multiplication used inside the division. As described in Section 5.1, even though methods that use the middle product have been proposed [56] [57], it is difficult to implement such algorithms efficiently using SIMD instructions. Therefore, a SIMD-instruction-friendly method, such as the proposed algorithm (Algorithm 8), which is a multiplication-based division method, should improve efficiency.

Another way to speed up division is to implement multiplication and division processes that are asymptotically faster, especially when the operand sizes are larger than those considered in this thesis. Since the computational complexity of division also depends on that of the multiplication used internally, future work on multiplication should also apply to division. The Newton–Raphson method is an effective division algorithm for large operands.

Since this division method is also multiplication-based, it has the potential to be processed at high speed using SIMD instructions.

Acknowledgments

I would like to express appreciation to my principal supervisor, Professor Daisuke Takahashi, who has provided me with a lot of support. He gave me much advice and many comments about my research and related matters. They helped me to progress smoothly in my research throughout the entire PhD course and I am deeply indebted to him.

I would also like to express appreciation to Professor Akira Nukada, who was my sub-supervisor. We also had helpful discussions, where he made a variety of useful comments, allowing me to improve the quality of my education and research.

I would like to express my special thanks to all my thesis committee members, Professor Daisuke Takahashi, Professor Akira Nukada, Professor Taisuke Boku, Associate Professor Takashi Nishide, and Associate Professor Yasunori Futamura, for reading the thesis and giving me helpful advice and comments.

I sincerely thank the members of the High Performance Computing Systems Laboratory in University of Tsukuba for various types of support.

This research in part used the computational resources of the Wisteria-O provided by the Multidisciplinary Cooperative Research Program in the Center for Computational Sciences, University of Tsukuba.

Finally, I would also like to express my deepest thanks to my family for their warm encouragement.

References

[1] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture. *IEEE MICRO*, Vol. 16, No. 4, pp. 42–50, 1996.

[2] Keith Diefendorff, Pradeep K Dubey, Ron Hochsprung, and HASH Scale. AltiVec extension to PowerPC accelerates media processing. *IEEE MICRO*, Vol. 20, No. 2, pp. 85–95, 2000.

[3] Srinivas K Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE MICRO*, Vol. 20, No. 4, pp. 47–57, 2000.

[4] Chris Lomont. Introduction to Intel Advanced Vector Extensions. Technical report, 2011.

[5] Guangyu Shi, Min Li, and Mikko Lipasti. Accelerating search and recognition workloads with SSE 4.2 string and text processing instructions. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 145–153. IEEE, 2011.

[6] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE MICRO*, Vol. 36, No. 2, pp. 34–46, 2016.

[7] Intel. Cryptography Processing with 3rd Gen Intel Xeon Scalable Processors. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/cryptography-processing-with-3rd-gen-intel-xeon-scalable-processors-19-may-2021.pdf, May 2021.

[8] Intel. Deep Learning with Intel AVX-512 and Intel Deep Learning Boost Tuning Guide on 3rd Generation Intel Xeon Scalable Processors. `https://www.intel.com/content/dam/develop/external/us/en/documents/Deep-Learning-with-Intel-AVX512-and-Intel-Deep-Learning-Boost-Tuning-Guide-on-3rd-Generation-Intel-Xeon-Scalable-Processors.pdf`, May 2021.

[9] Toshio Yoshida. Fujitsu high performance CPU for the Post-K Computer. In *Hot Chips*, Vol. 30, p. 22, 2018.

[10] Jinpil Lee, Francesco Petrogalli, Graham Hunter, and Mitsuhisa Sato. Extending OpenMP SIMD support for target specific code and application to ARM SVE. In *International Workshop on OpenMP*, pp. 62–74. Springer, 2017.

[11] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, Vol. 21, No. 2, pp. 120–126, 1978.

[12] Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pp. 417–426. Springer, 1985.

[13] Jinhu Li and Jeffrey S. Racine. Maxima: An Open Source Computer Algebra System. *Journal of Applied Econometrics*, Vol. 23, No. 4, pp. 515–523, 2008.

[14] Stephen Wolfram. *Mathematica: a system for doing mathematics by computer*. Addison Wesley Longman Publishing Co., Inc., 1991.

[15] Torbjörn Granlund. GNU Multiple Precision Arithmetic Library 6.2.1. `https://gmplib.org/`, Nov 2020.

[16] Wolfram. Third-Party Licenses GMP. `https://www.wolfram.com/legal/third-party-licenses/gmp.html`, 2022.

[17] Makoto Miyazaki and Susumu Matsumae. Improving Multiple Precision Integer Multiplication on GPUs. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, pp. 358–361. IEEE, 2017.

[18] Boon-Chiao Chang, Bok-Min Goi, Raphael C-W Phan, and Wai-Kong Lee. Accelerating Multiple Precision Multiplication in GPU with Kepler Architecture. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 844–851. IEEE, 2016.

[19] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. `https://cdrdv2.intel.com/v1/dl/getContent/671200`, June 2022.

[20] Arm. Arm Architecture Reference Manual. `https://developer.arm.com/documentation/ddi0487/ha/`, April 2022.

[21] Arm. ARM C Language Extensions for SVE. `https://developer.arm.com/documentation/100987/latest`, October 2020.

[22] Intel. Using Streaming SIMD Extensions (SSE2) to Perform Big Multiplications, version 2.0. Technical Report AP-941, No. 248606-001, 2000.

[23] Henry S. Warren. *Hacker's Delight*. Addison-Wesley, 2nd edition, 2012.

[24] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[25] Intel. Intel Xeon Phi Coprocessor Architecture Overview. `https://www.intel.com/content/dam/develop/external/us/en/documents/intel-c2-ae-xeon-phi-e2-84-a2-coprocessor-architecture-overview.pdf`, August 2013.

[26] Intel. Galois Field New Instructions (GFNI) Technology Guide. `https://builders.intel.com/docs/networkbuilders/galois-field-new-instructions-gfni-technology-guide.pdf`, July 2021.

[27] Intel. Intel Intrinsics Guide. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`, August 2022.

[28] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael

Premillieu, et al. The ARM scalable vector extension. *IEEE MICRO*, Vol. 37, No. 2, pp. 26–39, 2017.

[29] Shay Gueron and Vlad Krasnov. Software Implementation of Modular Exponentiation, Using Advanced Vector Instructions Architectures. In *Proc. International Workshop on the Arithmetic of Finite Fields (WAIFI 2012), Lecture Notes in Computer Science*, Vol. 7369, pp. 119–135. Springer, 2012.

[30] OpenSSL. `https://www.openssl.org/`.

[31] Pierre Fortin, Ambroise Fleury, François Lemaire, and Michael Monagan. High-performance SIMD modular arithmetic for polynomial evaluation. *Concurrency and Computation: Practice and Experience*, Vol. 33, No. 16, p. e6270, 2021.

[32] Anastasia Keliris and Michail Maniatakos. Investigating Large Integer Arithmetic on Intel Xeon Phi SIMD Extensions. In *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pp. 1–6. IEEE, 2014.

[33] Shay Gueron and Vlad Krasnov. Accelerating Big Integer Arithmetic Using Intel IFMA Extensions. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pp. 32–38. IEEE, 2016.

[34] Nir Drucker and Shay Gueron. Fast Modular Squaring with AVX512IFMA. In *16th International Conference on Information Technology-New Generations (ITNG 2019)*, pp. 3–8. Springer, 2019.

[35] Peter L. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, Vol. 44, pp. 519–521, 1985.

[36] Intel. 2nd Gen Intel Xeon Scalable Processors Specification Update. `https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/2nd-gen-xeon-scalable-spec-update.pdf`, July 2022.

[37] Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2010.

[38] Andi Kleen. A numa api for linux. *SUSE Labs*, August 2004.

[39] Anatolii. Karatsuba and Yu. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady*, Vol. 7, p. 595, 1963.

[40] Bérenger Bramas. A fast vectorized sorting implementation based on the ARM scalable vector extension (SVE). *PeerJ Computer Science*, Vol. 7, p. e769, 2021.

[41] Xiuwen Wan, Naijie Gu, and Junjie Su. Accelerating Level 2 BLAS Based on ARM SVE. In *2021 4th international conference on advanced electronic materials, computers and software engineering (AEMCSE)*, pp. 1018–1022. IEEE, 2021.

[42] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, Vol. 28, No. 2, pp. 135–151, 2002.

[43] Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, and Sergio Saponara. Fast deep neural networks for image processing using posits and ARM scalable vector extension. *Journal of Real-Time Image Processing*, Vol. 17, No. 3, pp. 759–771, 2020.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[46] Benjamin Buhrow, Barry Gilbert, and Clifton Haider. Parallel modular multiplication using 512-bit advanced vector instructions. *Journal of Cryptographic Engineering*, Vol. 12, No. 1, pp. 95–105, 2022.

[47] Fujitsu. A64FX Microarchitecture Manual. `https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.3.pdf`, October 2020.

[48] Andrei L Toom. The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. *Soviet Mathematics Doklady*, Vol. 3, pp. 714–716, 1963.

[49] Arm. Introduction to SVE2. `https://developer.arm.com/documentation/102340/0001/Introducing-SVE2`, May 2021.

[50] Donald E Knuth. *The Art of Computer Programming, volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, 1997.

[51] Tudor Jebelean. Practical Integer Division with Karatsuba Complexity. In *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pp. 339–341. ACM, 1997.

[52] Christoph Burnikel and Joachim Ziegler. Fast Recursive Division. *Research Report MPI-I-98-1-022*, 1998.

[53] Alan H Karp and Peter Markstein. High Precision Division and Square Root. *ACM Transactions on Mathematical Software*, Vol. 23, No. 4, pp. 561–589, 1997.

[54] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*, pp. 311–323. Springer, 1986.

[55] Niels Möller and Torbjorn Granlund. Improved Division by Invariant Integers. *IEEE Transactions on Computers*, Vol. 60, No. 2, pp. 165–175, 2011.

[56] David Harvey. The Karatsuba integer middle product. *Journal of Symbolic Computation*, Vol. 47, No. 8, pp. 954–967, 2012.

[57] William Bruce Hart. Efficient divide-and-conquer multiprecision integer division. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pp. 90–95. IEEE, 2015.

[58] Daisuke Takahashi. A Parallel Algorithm for Multiple-Precision Division by a Single-Precision Integer. In *Large-Scale Scientific Computing*, pp. 729–736. Springer, 2008.

[59] Niall Emmart and Charles Weems. Parallel multiple precision division by a single precision divisor. In *2011 18th International Conference on High Performance Computing*, pp. 1–9. IEEE, 2011.

[60] Torbjörn Granlund and Peter L Montgomery. Division by Invariant Integers using Multiplication. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pp. 61–72. ACM, 1994.

[61] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, Vol. 19, pp. 297–301, 1965.

Appendix A
List of Publications

Journal Paper

1. Takuya Edamatsu and Daisuke Takahashi: Fast Multiple-Precision Integer Division Using Intel AVX-512, IEEE Transactions on Emerging Topics in Computing, (in press).

Conference Papers

1. Takuya Edamatsu and Daisuke Takahashi: Acceleration of Large Integer Multiplication with Intel AVX-512 Instructions, Proc. 20th IEEE International Conference on High Performance Computing and Communications (HPCC-2018), pp. 211-218 (2018).

2. Takuya Edamatsu and Daisuke Takahashi: Accelerating Large Integer Multiplication Using Intel AVX-512IFMA, Proc. 19th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2019), Part I, Lecture Notes in Computer Science, Vol. 11944, pp. 60-74, Springer (2020).

3. Takuya Edamatsu and Daisuke Takahashi: Efficient Large Integer Multiplication with Arm SVE Instructions, Proc. International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2023), pp. 9-17 (2023).